Cooperative Control of Autonomous Vehicles

Thesis
Ph.D Course on:
Automation, Robotics and Bioengineering
Cycle XX (2005)

Tutor:
Prof. Mario Innocenti
Student:
Andrea Bracci

UNIVERSITÀ DI PISA

University of Pisa
Department of Electrical Systems and Automation

Contents

CONTENTS

Introduction

Cooperative control is a relatively new research field and it has gained increasing interest due to the progress in microelectronics and computer technology that made possible the realization of complex control laws [1, 2].

The main feature of Cooperative Control of interest in this work is the presence of many (at least two) independent agents that interact with each other and with the environment where they operate. A key issue related to the above is the need of a communication link between the agents, since they must share some information about their internal state and the environment. One of the main reasons that fosters the research in cooperative control is the added advantage whenever a mission is accomplished by a team of agents instead of a single agent. Common applications of cooperative control involves both civilian and military operations such as, for instance, area surveillance [3], enemy targets detection [4, 5, 6, 7], targets attack [8].

In the context of cooperative control the agents are autonomous vehicles (typically referred as *Unmanned Autonomous Vehicles*, UAVs) having a certain degree of autonomy that allows them to indepen-

dently make decisions and accomplish different tasks. The term *autonomous* means that a human is neither onboard nor remotely commands the vehicle. For this reason UAVs have a great potential accomplishing missions otherwise impossible or too risky for humans. There are many possible applications for the use of unmanned vehicles. For instance, a heavily contaminated environment could not be explored by humans because of the tremendous consequences that the contamination could produce on them; however an UAV could. Another example is space exploration, where no other option but autonomous systems is foreseen in the near and the medium term.

Remote control of an unmanned vehicle could be effective in many cases but there can be the problem of rapidly reacting to the environmental changes that could be delayed along the transmission channel. For this reason a completely autonomous (and hence not remotely controlled) vehicle may be a better choice.

The problems encountered in cooperative control of unmanned vehicles are several: since the UAVs must move inside an environment that could contain obstacles, one objective is to produce obstacle-free paths that the vehicles can follow in order to accomplish their tasks. In addition, those obstacles may suddenly appear as mission evolves (*pop-up obstacles*) thus producing a much more risky scenario. The problem of path-planning among obstacles is well known and it has been well studied. In [9], Kapoor *et. al.* propose an efficient algorithm to compute the Euclidean shortest path inside a polygon with several polygonal obstacles. An improvement of the result of Kapoor can be found in [10]: in the reference paper, Hershberger proposes an efficient wavefront propagation to obtain an optimal algorithm which is capable of producing the shortest path

among obstacles in the plane. The algorithm in [10] is optimal since it is proven to have the lowest possible time complexity.

Another approach to path-planning can be found in[11], where an artificial potential field approach has been proposed. The procedure consists in building a potential field around the obstacles and use it to compute a safe path.

A visibility graph approach is also studied in literature [12, 13]. In the paper of Huang a dynamic version of the visibility graph has been proposed in the case of convex obstacles only.

All the cited algorithms are capable of producing the shortest obstacle-free path and, with the exception of [13], are not dynamic. A different approach to path-planning is based on a sort of tessellation. Both Voronoi tessellation [14] and Delaunay triangulation [15] have been studied. The *Constrained Delaunay Triangulation* (CDT) has been widely employed since the resulting triangles do not overlap and a fully dynamic version has been developed [16, 17]. An application of the CDT for path-planning purposes can be found in [18]. Though not optimal, path-planning in triangulations has the great advantage of being dynamic and hence more favorable for real-time purposes.

Vehicles capabilities can be directly dealt in the path-planning process taking into account, for instance, the minimum turning radius. Bullo *et. al.* [19, 20] propose a procedure to build a path composed by straight line and circular arcs with a minimum radius value. By this way the obtained trajectory can be directly followed by a Dubins' vehicle.

Obstacle avoidance is not necessarily a cooperative control problem if vehicles collisions are neglected. In this case, in fact, each ve-

hicle can independently produce its own obstacle-free path without regard of the others, and the issue of collision avoidance among the team may be considered next. Direct consideration of this problem is addressed in literature, see for instance [21, 22] where a decentralized approach is proposed.

A more *cooperative* problem is the task accomplishment, since in this case the UAVs must act as a team in order to achieve a common objective. This problem directly involves the intra-team communications because the cooperation requires some sort of data exchange between the agents of the team. The amount and the type of data to be exchanged can vary depending on the chosen control algorithm.

The optimal solution of the task assignment problem is *NP-Hard* and hence it is computationally intensive even for small-sized scenarios. For this reason suboptimal procedures have been studied, and their solution can be compared in terms of residual cost [23]. In [24] an *Hungarian Algorithm*-based procedure is proposed. The *Hungarian Algorithm* [25, 26] is a simple and fast task-allocation technique and a distributed version is also available [27].

An alternative technique is based on the *Auctioning* [28]. This technique consists in simulating many auctions (one for each task) among the team letting the vehicles following predefined dynamics (*auctioneer* or *bidder*). The *Auctioning* procedure is decentralized and allows dynamic reassignments during the mission.

Path-planning and task-assignment can be merged into a unique procedure formulating them as a *Mixed Integer Linear Programming* problem [29, 30]. This procedure has the drawback of a high computational load even in a receding horizon realization [31].

CONTENTS

The general cooperative control problem is very complex and involves many issues. In this work many assumptions are made in order to simplify the treatment. In particular, it is assumed that the obstacles are represented by polygons in the plane and the vehicles are dimensionless points. The overall dimension of the vehicles is taken into account by enlarging the real obstacles such that, if the point corresponding to a vehicle lies on the edge of an obstacle, then there is no collision. Sensor measurement issues, and data-link hardware configuration are neglected.

The main contribution of the thesis is the development of dynamic and fast strategies of mission management, such that a real-time cooperative control implementation can be easily employed. Many existing procedures are very effective in the sense that a cost index is minimized or some properties are assured; however the not-dynamism of such procedure prevents the implementation even in small-sized scenarios. This is the reason why a dynamic procedure is sought. The techniques developed are compared in terms of final result (that is, the final value of a predefined cost index) and in terms of the total computational load. Although hardware limitations of real-time implementation are not directly taken into account, these are always kept in mind in the treatment considering the total amount of computations and data to be transmitted.

Chapter 1

Problem Statement

In this chapter the general problem of cooperative control is stated. A team of unmanned vehicles moves into a real 3-D environment (typically a portion of the Earth) and it must accomplish some given mission. There can be several types of missions, each of these with different requirements. The objective of the vehicles of the team is to cooperate in order to complete the mission while maximizing the benefits, or minimizing the costs. Typically, the objectives are represented by a various number of tasks to be accomplished, and each task may be characterized by reaching a particular position (the target point) and, in many cases, performing a predetermined action. A common threat for the team is the presence of obstacles and dangerous zones that must be avoided at every time. Threats can be either fixed or moving, thus increasing the complexity of the scenario itself. Another subtle threat is represented by the potential collision between the vehicles; this problem is not considered in the present thesis, and details can be found in [32, 33, 21].

Although not directly dealt in this work, communication limitations between the vehicles play also an important role because cooperation requires some data exchange and then effective and efficient communication links are necessary. Some issues on the communication between the vehicles of a team can be found in [34]. In the present thesis this problem is treated to some extent by focusing on the amount of data that must be sent throughout the team.

The chapter is organized as follows: in the first section the *scenario* is defined, focusing on the different objects that characterize it. Then the problem due to the presence of dynamic objects is presented and next the relations between the tasks and the targets are shown. Finally the statement of the general cooperative control problem is presented.

1.1 Scenario Definition

The scenario $\mathbb{S}$ is a set of interacting objects having different properties and it is defined as a tuple as in equation (1.1)

$$\mathbb{S} = \{E, V, T, O\} \tag{1.1}$$

where E is the environment, V is the set of vehicles, T is the set of targets and O is the set of obstacles. These objects have different characteristics and dynamics; the next subsections clarify the role and the properties of each.

1.1.1 Environment

The environment is the space where all the objects are contained. In the general case it is assumed that the environment is a portion of the Earth and then the generic environment is a 3-D space. However a common approach is to consider a 2-D (flat) environment, as all the objects can move on a plane. This simplification is due to the fact that the distances between the vehicles and the targets, in terms of altitude, are typically negligible with respect to the distance measured on the plane parallel to the Earth at a given point. However, since some vehicles can move along the three dimensions (air and underwater vehicles), the common approach is to plan the mission in a 2-D space and then to introduce the third dimension for the vehicles that can move along it. Clearly, the altitude position of a target becomes very important whenever a vehicle must visit it.

A typical cooperative mission evolves in a bounded domain. Without loss of generality, the domain is assumed to be rectangular and defined by its four corners $(x_{min}, x_{max}, y_{min}, y_{max})$. The environment E is then correctly defined as in (1.2):

$$E = \{x, y \in \Re^2 | x_{min} \leq x \leq x_{max}; y_{min} \leq y \leq y_{max}\} \tag{1.2}$$

The values of the bounds of the environment are arbitrary and do not affect the cooperative algorithms, hence these bounds can be set large enough to avoid that one or more objects lie outside of them.

1.1.2 Vehicles

The set of vehicles V in (1.1) is defined in (1.3), where V_i is a generic vehicle and N_{veh} is the total number of vehicles in $\mathbb{S}$

$$V = \bigcup \{V_i\} \qquad i = 1 \dots N_{veh} \tag{1.3}$$

The vehicles are the active and controllable objects of the scenario. Each vehicle V_i is characterized by its own inner loop dynamics that in the general case can be written in the standard nonlinear form as in (1.4).

$$\dot{x}_{IL,i} = f_i(x_{IL,i}, u_{IL,i}) \tag{1.4}$$

where $x_{IL,i}$ represents the i^{th} vehicle inner loop state variable. An outer kinematic controller is frequently used in order to achieve a desired position and velocity behavior. The kinematics to be controlled can be written as in (1.5).

$$\dot{x}_{K,i} = f_i(x_{K,i}, u_{K,i}) \tag{1.5}$$

where $x_{K,i}$ represents the i^{th} vehicle kinematic variable. The cooperative controller is a higher level controller and it is generally placed outside the kinematic loop. The role of the cooperative controller is to interact with the scenario $\mathbb{S}$ and to generate the correct references to the kinematic loop. The references are typically the waypoints of the desired trajectory, or some similar path-related variables.

It is common practice in cooperative control to neglect the inner dynamics and the kinematics loop, assuming that suitable controllers have already been introduced, if necessary. However there

exist cooperative control procedures (such as MILP-based cooperative control, see chapter 2) that are capable of directly deal with the inner dynamics.

1.1.3 Targets

The set of targets T in (1.1) is defined in (1.6) where T_j is a generic target, and N_{tar} is the total number of targets in $\mathbb{S}$.

$$T = \bigcup \{T_j\} \qquad j = 1 \dots N_{tar} \tag{1.6}$$

A target T_j is identified by its xy-coordinates in the plane x_{Tj}, y_{Tj} and it is assumed to be a dimensionless point. The target position is sensed by the vehicles during the mission and hence, since in general it is not known *a priori*, the value of N_{tar} can vary during the mission.

1.1.4 Obstacles

The set O in (1.1) represents the obstacles and it is defined in the following:

$$O = \bigcup \{O_k\} \qquad k = 1 \dots N_{obs} \tag{1.7}$$

where each obstacle O_k is considered as a polygon in the plane and it is identified by the coordinates of its vertices as follows:

$$O_k = \bigcup \{(x_{i,k}, y_{i,k})\} \qquad i = 1 \dots N_{vert,k} \tag{1.8}$$

The *real* obstacles (represented by the internal points of the polygons) are some forbidden areas where the vehicles are not allowed

to enter. This areas can be either physical obstacles such as mountains or buildings, and/or dangerous zones such as fires and radar-controlled areas. In all the cases it is assumed that if a vehicle enters an obstacle then it will be lost, hence the first requirement of the mission is that each vehicle avoid obstacles at all times.

1.2 Static and Dynamic Objects

In section 1.1 the scenario objects have been defined neglecting the potential dynamism of both targets and obstacles. However, in real environments, these objects can move and may become harmful to the vehicles. In addition, if a target moves, the vehicles must replan their routes in order to take into account the scenario modification. In this section the effects of the presence of such dynamic objects are shown in terms of control requirements.

The static objects can be either fixed targets or fixed obstacles. Such objects are relatively easy to deal with because their position is fixed and typically known *a priori* and hence off-line control procedures can be used.

A more challenging control problem is dealing with dynamic objects, that is, objects that change their position and their behavior as the mission evolves. Such objects may be represented by moving targets, moving obstacles, *pop-up targets* and *pop-up obstacles*. The presence of dynamic objects makes the cooperative control problem much more complex since off-line computations become useless and dynamic on-line algorithms are required. Moreover, active sensing must be employed in order to recognize scenario variations. In the

remaining of this work the term ***dynamic*** procedure or ***dynamic*** algorithm denotes the capability of such procedure (or algorithm) to take into account scenario variations avoiding re-run of the entire procedure from the beginning. It is clear that dynamic procedures are more feasible for real-time purposes since typically require less computational load.

It will be more clear later that the dynamic and unknown objects are the main reason for continuous research in cooperative control because, in the presence of static objects only, the resulting problem could be efficiently solved off-line obtaining optimal solutions.

1.3 Targets and Tasks

In subsection 1.1.3 the targets are defined as dimensionless points in the plane neglecting what a vehicle has to do once it has reached a given target. For this reason it is necessary to introduce the concept of *task*:

Definition 1 *A task is a predefined action that a vehicle must perform whenever it visits a target.*

A target may have many joint tasks that can be accomplished by one or more vehicles simultaneously or sequentially. It is important to notice that the accomplishment of a task may require a particular trajectory to be followed by the vehicle. Each task requires a cost to be accomplished and its cost strictly depends on the joint target position and on the task type. However, the cost of each task depends not only on the task itself but also on the vehicle that could

accomplish it. In fact, in the scenario there are many vehicles with different positions and capabilities and hence different vehicles may have different costs in accomplishing a given task. In the general case, a task cost is given by the combination of the distance between the vehicle and the joint target, and the task cost itself that depends on the task type. In the remaining of this subsection the main tasks considered are presented.

Reconnaissance. The *reconnaissance task* Υ_{RE} requires that a vehicle searches for potential targets in a given area. For this reason the vehicle assigned to this task must follow a predefined path in order to cover the entire area at least once (see for instance [35, 5]).

Visit. The *visit task* Υ_{VI} is the simplest one because it only requires that the vehicle *passes on* the joint target.

Attack. The *attack task* Υ_{AT} is similar to the visit task with the only difference that the attack action may require a given time to be completely accomplished. Under the name *attack* many actions are included such as fire extinguishment and survivor rescue; all of these tasks must be performed at the location of the joint target and the vehicles need not to follow a predefined trajectory as in the *reconnaissance task*. However, considering for instance a small unmanned airplane, it is not possible to keep still over the target point and then at least a circular path must be followed.

Verify. The *verify task* Υ_{VE} can be performed after an attack task and it requires that a vehicle goes to the joint target and verifies

that the *attack task* has been successfully accomplished.

Track. The *track task* Υ_{TR} requires that a vehicle tracks a given target for a given time. This task is present whenever a target is moving and its position must be known at every time. Since target tracking can be employed using a radar, this type of task does not require that the vehicle visit the specific target, but it is sufficient to maintain the distance lower than some specified minimum.

1.4 Cooperative Control Problem

Let $\mathbb{S}$ be a given scenario with N_{veh} vehicles, N_{tar} targets and N_{obs} obstacles. Assume that there exist targets and obstacles that are not known *a priori* and that their knowledge can be obtained during the mission. Define a set of tasks Υ_j for each target T_j as in (1.9) considering tasks precedences.

$$\Upsilon_j = \bigcup \{\Upsilon_{j,x}\} \qquad x \in \{RE, VI, AT, VE, TR\} \tag{1.9}$$

Let $c_{i,j,k}$ be the cost that the i^{th} vehicle has to accomplish task $\Upsilon_{j,k}$. With the previous statements we are now ready to define a generic cost index J depending on the scenario $\mathbb{S}$ and the set of tasks Υ.

$$J = f(S, \Upsilon) = f\left(\bigcup \{c_{i,j,k}\}\right) \tag{1.10}$$

where:

$$\Upsilon = \bigcup \{\Upsilon_j\} \tag{1.11}$$

The cooperative control problem can then be formulated as follows:

Problem 2 *Determine obstacle-free trajectories for each vehicle such that all the tasks in* Υ *are accomplished minimizing the team-cost index* J *defined in (1.10).*

The main property of the stated problem is that the cost index to be minimized includes all the vehicles in the team. The general solution of problem (2) is not known in closed form and then numerical solvers are required. In the next chapters many different approaches are presented. The first approach is based on MILP and it is capable of producing the optimal solution with the drawback of a very large computational load. Other approaches are based on the separation of the trajectory-generation and the task-assignment phases. Those methodologies are not capable of producing the optimal solution in all the cases but the computational load is sensibly reduced and real-time applications can be realized.

Chapter 2

MILP-Based Cooperative Control

Mixed Integer Linear Programming (MILP) problems are a particular version of Linear Programming (LP) problems, in that they involve both real and integer variables. The cost index to be minimized is a linear combination of the optimization variables and it can reflect different minimization objectives. The exact solution of a MILP problem can be numerically found using dedicated solvers such as CPLEX, *Mathematica* and *lpsolve*. Many problems of cooperative control can be formulated in MILP form including obstacle avoidance, minimum fuel consumption, collision avoidance and tasks precedences. The mission objectives are typically considered in the cost index to be minimized, while the other issues are dealt in the constraints definition. MILP approach to cooperative control is very powerful in the sense that it allows the treatment of many problems, but the main drawbacks are the intrinsic centralization and

the not dynamism of the entire procedure. Moreover the total computational load may become too high even for small sized scenarios. These disadvantages limit the use of MILP-based cooperative control in real-time implementation.

In this chapter the main properties of MILP-based cooperative control are reviewed, starting from the statement of a general MILP problem and specializing it into a cooperative control application. Next some numerical considerations are made. Some concluding remarks end the chapter.

2.1 Mixed Integer Linear Programming Problems

Consider a generic cost index J defined as in (2.1)

$$J = \sum_{i=1}^{N} q_i x_i = \mathbf{q}^T \mathbf{x} \tag{2.1}$$

where the vectors $\mathbf{q}$ and $\mathbf{x}$ contain some weights and the optimization variables respectively and N is an integer positive value. Vector $\mathbf{x}$, whose length is n_{tot} contains both real and integer variables:

$$x_i \in \begin{cases} \mathbb{R} & \text{if} \quad i = 1 \dots n_{real} \\ \mathbb{N} & \text{if} \quad n_{real} < i \leq n_{tot} \end{cases} \tag{2.2}$$

Consider then a generic linear inequality constraints set involving the variables x_i as in (2.3)

$$A\mathbf{x} \leq \mathbf{b} \tag{2.3}$$

The MILP problem can then be formulated as follows:

Problem 3 (Mixed-Integer Linear Programming Problem) *Minimize the cost index J in (2.1) subject to the constraints (2.3) involving both real and integer variables as in (2.2).*

If the variables involved in the optimization are real only then the problem reduces to a Linear Programming one, and the optimal solution can be efficiently found by standard solvers. The presence of both real and integer variables causes the problem to become much harder to be solved. In fact dedicated and more complex solvers must be employed (such as CPLEX, *Mathematica* and *lpsolve*). Finally, many optimization problems encountered in cooperative control involves binary variables (instead of more general integers variables). In this thesis such problems will be always referred as MILP problems.

2.2 MILP in Cooperative Control

In this section the MILP formulation of many issues encountered in cooperative control problem 2 is briefly presented following [29]. Consider for the moment the presence of a single vehicle and define a generic continuous-time cost index J as follows:

$$J = \int_0^\infty \left(\mathbf{q}^T|\mathbf{s}| + \mathbf{r}^T|\mathbf{u}|\right) dt \tag{2.4}$$

where $\mathbf{q}$ and $\mathbf{r}$ are two nonnegative weighting vectors; $\mathbf{s}$ and $\mathbf{u}$ are the vehicle state-variables and inputs respectively. In order to numerically solve the optimization problem, the cost-index in (2.4) must

be discretized and a finite time-horizon T[1] must be set. This time horizon must be divided into $N = T/\triangle t$ steps where $\triangle t$ is the chosen sample time. Finally a terminal cost $f(s_N)$ term must be added in order to consider the remaining time $T \to \infty$. The cost index is thus modified as follows:

$$\min_{s_i,u_i} J_T = \min_{s_i,u_i} \sum_{i=0}^{N-1} \left(\mathbf{q}^T|\mathbf{s}_i| + \mathbf{r}^T|\mathbf{u}_i|\right) \triangle t + f(s_N) \tag{2.5}$$

subject to:

$$s_{i+1} = \mathbf{A}s_i + \mathbf{B}u_i \tag{2.6}$$

where the two matrices $\mathbf{A}$ and $\mathbf{B}$ represent the linearized discrete dynamics of the vehicle.

Dividing by $\triangle t$, considering $f(s_N) = p^T s_N$ and introducing some positive slack-variables w_i and v_i the problem becomes

$$\begin{aligned} \min_{w_i,v_i} J_T = \min_{w_i,v_i} & \sum_{i=0}^{N-1} \mathbf{q}^T\mathbf{w}_i + \sum_{i=0}^{N-1} \mathbf{r}^T\mathbf{v}_i + p^T\mathbf{w}_N \\ \texttt{subject to} \quad & s_{ij} \le w_{ij} \\ & -s_{ij} \le w_{ij} \\ & u_{ik} \le v_{ik} \\ & -u_{ik} \le v_{ik} \\ & \mathbf{s}_{i+1} = \mathbf{A}\mathbf{s}_i + \mathbf{B}\mathbf{u}_i \end{aligned} \tag{2.7}$$

where the indices j and k denote the components of the state and input vector respectively.

[1]This symbol T must not be confused with the *targets set* of chapter 1

In the case of multi-vehicle scenarios, the cost index must be modified as follows:

$$\begin{aligned}
\min_{w_i,v_i} J_T = \min_{w_i,v_i} & \sum_{p=1}^{N_{veh}} \left(\sum_{i=0}^{N-1} \mathbf{q}^T \mathbf{w}_{pi} + \sum_{i=0}^{N-1} \mathbf{r}^T \mathbf{v}_{pi} + p^T \mathbf{w}_{pN} \right) \\
\texttt{subject to} \quad & s_{pij} - s_{pf} \leq w_{pij} \\
& -s_{pij} + s_{pf} \leq w_{pij} \\
& u_{pik} \leq v_{pik} \\
& -u_{pik} \leq v_{pik} \\
& \mathbf{s}_{p,i+1} = \mathbf{A}\mathbf{s}_{pi} + \mathbf{B}\mathbf{u}_{pi}
\end{aligned} \tag{2.8}$$

where the values s_{pf} represent the desired final state of the vehicles. Additional constraints can be established depending on the system requirements. It is possible to take into account the presence of both static and dynamic obstacles [29, 36]. Nonetheless, in the last case the obstacles dynamism is supposed to be completely known *a priori* and hence it is not a realistic case. Collision avoidance and task precedence issues can be directly included in MILP formulation using additional integer variables [29]. The resulting constraints can be set in the form (2.3).

2.3 Solving MILP

Once the cooperative control problem is formulated into a MILP form, the exact solution can be numerically found using dedicated solvers. The problem complexity depends on the scenario size, on the constraints and the optimization time-horizon. In real situations

the number of optimization variables can be very large thus resulting in a very complex problem.

Once the time horizon is fixed, the number of discretization steps N must be chosen large enough to get a sufficiently fine control law, however, this value must be sufficiently small in order to obtain a more computationally feasible problem. As scenario size grows, this issue becomes much more critical because the problem may become numerically intractable. A *receding horizon* approach can be useful in order to reduce the computational load of the entire procedure, at the expense of a globally sub-optimal solution. Anyhow the computational time of every time step of the receding horizon control may become too long then resulting into a not real-time feasible control law.

Example 4 *As an example, consider the scenario represented in figure 2.1(see figure 2 in [29]).*

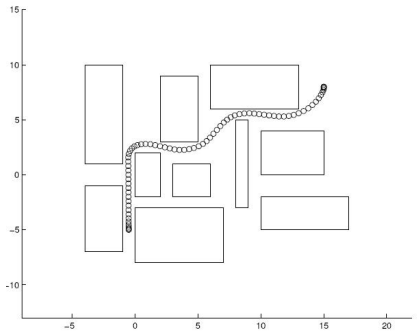

Figure 2.1: Sample scenario

In Table 2.1 many results for different time horizons are pre-

$T_{hor}(s)$	$T_{arr}(s)$	E_{tot}	$T_{comp}(s)$	$T_{it}(s)$
3.0	9.2	35.7	129	1.40
3.5	9.0	34.1	267	2.97
4.0	8.4	31.6	441	5.25
4.5	8.2	30.4	728	8.88
5.0	8.2	30.4	1213	14.79

Table 2.1: Computational time for many time horizons [29]

sented:

T_{hor} *Time horizon*

T_{arr} *Arrival time*

E_{tot} *Fuel consumption index*

T_{comp} *Total computational time*

T_{it} *Single iteration computational time*

It is clear that even using a receding horizon *approach the computational time of the entire mission is greater than the real mission time, thus resulting into an intrinsic non-real-time controller. Clearly the* receding horizon *improves the computational time, however this is a very simple scenario involving one vehicle and one target only.*

2.4 Concluding Remarks

Chapter 3

Path Planning

The chapter addresses the problem of finding the shortest path between two points in the plane avoiding many holes. Though not a proper team-optimization problem, this is a crucial issue in cooperative control, because the vehicles must move in a scenario with many obstacles and potentially dangerous zones. In addition, both obstacles and dangerous zones can move in an unknown way and a fast path-planning procedure may be necessary in order to take into account such variations. At this point, real vehicle capabilities (such as, for instance, the *minimum turning radius*) are neglected and the simple point-to-point shortest path is computed. A way to deal with the vehicles motion limitations can be found, for instance, in [19, 20].

The chapter is organized as follows: the general path planning problem is stated in section 3.1, then two optimal path-planning algorithms based on the *visibility graph* and the *Euclidean shortest path* are briefly presented. Next a set of triangulation-based sub-

optimal algorithms are presented focusing on the novel *adjacency-path* method which is very suitable for real-time purposes. Finally some concluding remarks end the chapter.

3.1 Problem Definition

Consider a closed and not self-intersecting polygon P_E in the plane and consider many other not self-intersecting polygons ($P_{H,i}$) representing many *holes* completely inside P_E. Consider then two points $\mathbf{p}_s$ and $\mathbf{p}_e$ inside P_E but outside the $P_{H,i}$. The general problem of path-planning can then be stated as follows

Problem 5 (Path-Planning Problem) *Determine the shortest obstacle-free path between two points* $\mathbf{p}_s$ *and* $\mathbf{p}_e$ *inside the polygon* P_E.

From a cooperative control point of view it is worth noticing that the solution of the previous problem gives the shortest obstacle-free path between each pair vehicle/target and target/target in the scenario. This is a nice property, since it is possible to concentrate the efforts on finding a fast procedure to determine the shortest obstacle-free path between two generic points and then to apply the chosen procedure many times in order to get the desired shortest paths between any pair of points in the plane.

The exact solution of the path-planning problem 5 is very important in the view of minimizing the fuel consumption of the vehicles, however, in real-time implementations, other issues must be taken into account:

- The obstacles can move during the mission and then the optimal solution could not be optimal after any longer.

- Due to the dynamic nature of the scenario the path-planning algorithm must be run many times during the mission
- The maximum update rate of the path is limited by the computational load of the path-planning algorithm

The objective of this chapter is then not only to find the shortest obstacle-free path between two points, but also to find a numerically fast and dynamic procedure in order to achieve a faster sampling rate. As it will become clear later, the known optimal algorithms are not dynamic and hence a complete *re-run* of the algorithm must be employed. At the end of the chapter a new dynamic triangulation-based sub-optimal algorithm is presented focusing with its advantages and drawbacks.

3.2 Optimal Algorithms

Two optimal algorithms that solve the shortest-path problem are presented in this section. It is important to notice that, except for particular cases in which there are two or more different but equally expensive paths, the optimal solution is only one and then both algorithms produce the same solution. The input data of the algorithms are the same, that is:

- The coordinates of $\mathbf{p}_s$ and $\mathbf{p}_e$ are completely known.
- The coordinates of the vertices of the polygons are completely known
- Both points are inside P_E.

- Both points are not inside any hole.
- A generic path cost is represented by its length.

3.2.1 Visibility Graph Based Algorithm

This algorithm is based on the definition of the so-called *Visibility Graph* ($\mathcal{VG}$) that is a graph whose nodes $\mathcal{N}_{VG}$ represent both the obstacles vertices and the two points $\mathbf{p}_s$ and $\mathbf{p}_e$, while the edges $\mathcal{E}_{VG}$ represent the set of feasible paths among the nodes. A feasible path between two points is a straight segment that does not cross any holes. The $\mathcal{VG}$ can be defined as follows:

$$\mathcal{VG} = \{\mathcal{N}_{VG}, \mathcal{E}_{VG}\} \tag{3.1}$$

where:

$$\mathcal{E}_{VG} = \{e_{i,j}, \forall i, j = 1 \dots n, |\overline{p_i p_j} \text{ does not cross any obstacle.}\} \tag{3.2}$$

Once the *visibility graph* has been correctly defined the shortest-path is found employing the *Dijkstra Algorithm* ($\mathcal{DA}$) between the start node and the end node. The $\mathcal{DA}$ is a well known graph-optimization routine and it is proven that, employing a Fibonacci heap, it runs in $\boldsymbol{O}(\|\mathcal{E}\| + \|\mathcal{N}\| \log \|\mathcal{N}\|)$ time. The result of the $\mathcal{DA}$ is the cheapest directed route among the graph nodes that links the start node with the end node. From a graph-optimization point of view, the shortest route between two nodes N_1 and N_2 is not necessarily the inverse of the shortest route between N_2 and N_1 and hence it is important to specify that the obtained route is oriented.

3.2 Optimal Algorithms

Following the definition of $\mathcal{VG}$ and using the result of the application of $\mathcal{DA}$, the obtained route represents the shortest obstacle-free path between the start point and the end point in P_E.

The computational load of the entire procedure depends both on the graph construction and the $\mathcal{DA}$. The construction time of the *Visibility Graph* is $\boldsymbol{O}(\|\mathcal{N}_{VG}\|^2)$ (see [12, 37] for many details), and then, since the number of admissible edges is typically lower than $\|\mathcal{N}_{VG}\|^2$, the computational load of $\mathcal{DA}$ is negligible with respect to the construction of $\mathcal{VG}$. It follows that the resulting computational load is globally $\boldsymbol{O}(\|\mathcal{N}_{VG}\|^2)$.

It is worth noticing that this path-planning procedure is capable of producing not only the shortest path, but also a more general *cheapest-path* as well. This fact can be used in cooperative control in order to introduce paths with different levels of risk.

Properties

The $\mathcal{VG}$-based path-planning is not a dynamic procedure since both $\mathcal{VG}$ and $\mathcal{DA}$ are not dynamic at all. However this is not a problem since its computational cost is not too high. The bottleneck of the entire procedure is given by the construction of the visibility graph that is time-consuming and, whenever scenario changes, must be run again from the beginning. Recently Huang *et al.* [13] have proposed a dynamic version of the $\mathcal{VG}$-based path planning but in the case of convex holes only.

As the number of holes grows, it is clear that the $\mathcal{VG}$ complexity increases and may become too complex or even redundant because of the presence of too many *close* or overlapped feasible edges. In

figure 3.1, for instance, there are many overlapped edges between the nodes at the top of the three obstacles thus resulting in a more complex visibility graph. However these redundant edges cannot be deleted since it is not known *a priori* which nodes the shortest route will visit and then all possible admissible arcs must be considered.

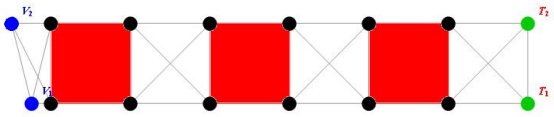

Figure 3.1: Redundant Visibility Graph

Finally, the visibility graph approach to path planning takes care of the whole scenario even if the resulting path is *close* few obstacles only.

3.2.2 Optimal Euclidean Shortest Path Algorithm

The computational cost of the *visibility graph* based path planning is sensibly larger than the theoretic lower bound of $\boldsymbol{O}(n \log n)$ (where n is the total number of vertices, including the holes, in the polygon P_E). This lower bound can be achieved following the procedure proposed by Hershberger [10]. This procedure is capable of finding the optimal *Euclidean Shortest Path* inside a polygon with many holes in the plane.

The algorithm consists in an efficient wave-propagation simulation, starting from the point $\mathbf{p}_s$. It is assumed that the wave propagates with a constant speed, and hence, the wavefront at time t

consists of all points in the plane whose shortest path to $\mathbf{p}_s$ is t. The boundary of the wavefront is a set of cycles, each composed by a circular arc. Each arc is generated by an obstacle vertex which is already covered by the wavefront (see Figure 3.2. The main problem in the algorithm is to deal with arcs intersections that can result into either a straight line or a hyperbola. The problem is efficiently solved employing a subdivision of the plane and an approximate wavefront. The details of the algorithm can be found in the reference paper [10].

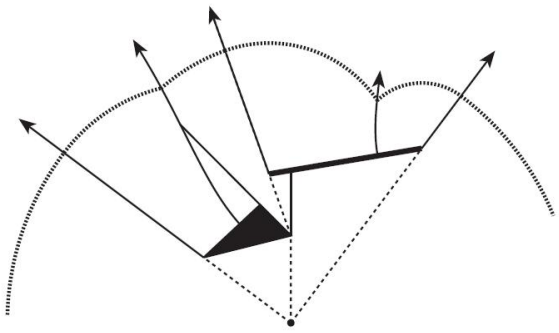

Figure 3.2: Simulated wavefront propagation of the Hershberger algorithm

Though the present procedure has a low computational cost, it requires to be re-run from the beginning as scenario changes, since it is not dynamic. For this reason different approaches to the path-planning problem are sought in order to reduce the required on-line computations.

3.3 Suboptimal Algorithms

The optimal path-planning algorithms presented in the previous section are not dynamic, and they must be run again whenever the scenario changes. This is not a desirable property in real-time applications. In this section, many suboptimal algorithms are presented focusing on the total computational load and on the dynamism of each. In general there is any *a priori* optimality degree guarantee but the dynamism of the procedure is preferred to the optimality. Moreover as stated, in section 1.1, the scenario dynamism is not known *a priori* and then the contingent optimality of the solution could lose its property even at the next time step. The main objective of this section is to determine an algorithm, which is capable of dealing with dynamic scenarios while requiring a low computational load.

3.3.1 Uniform Tessellation Based Algorithms

The first idea in reducing the path-planning complexity is to use a fixed tessellation of the scenario, such that the main computational load can be carried out off-line. The term *uniform* reflects the fact the tessellating elements are equally-shaped, although different element size is allowed. Unless explicitly stated, it is assumed that the resulting path is along the edges of the tessellating elements.

In order to obtain a correct tessellation the following step must be followed:

1. Choose a type of polygon (i. e. triangles, rectangles...).
2. Find the optimal shape of the tessellating polygon.

3. Determine the optimal size of the polygons.

4. Deal with irregular obstacles in the scenario.

In the remaining, all the previous issues will be considered.

Since the scenario is represented by a plane, the choice of the tessellating element is made by the 2D polygons. The simplest polygon is the triangle. It is the only polygon in which a vertex is linked to all the other vertices, and it is convex. These nice properties make the most common choice. Moreover, the previous properties also allow to state that there can not be neglected internal path inside each triangle.

Consider now an obstacle-free scenario and bound it into a rectangle defined as in section 1.1.1, and suppose that it is tessellated with equal triangles. The optimal shape of the triangle is found by minimizing the maximum error between the straight line linking two generic triangles vertices and the shortest path along the triangulation. Not surprisingly the optimal shape, found by numerical solution of the stated problem, is the equilateral triangle. Moreover the relative error does not depend on the triangles dimensions. This fact allow the use large triangulations without degrading the path-planning performance.

The size of the selected equilateral triangles depends on the typically irregular obstacles distribution. This fact prevents the direct use of an uniform tessellation. In fact, the triangle edges that cross one obstacle must be deleted and hence a finer tessellation must be employed in the neighborhood of these zones. Possible ways of dealing with this fact are presented next:

- The obstacles edges is subdivided into many segments and scalene triangles are built among these vertices and the vertices of the regular tessellation. This approach, though capable of recovering the entire obstacles edge, is not suitable for real-time purposes because the scenario dynamism prevents the application of heavy off-line computations.

- The not-admissible triangles edges are deleted and the equilateral triangles are refined (i. e. by a factor 2^n) in the neighborhood of the obstacles. This approach recovers the same considerations about the maximum relative error between the optimal path and the approximate one. However, on-line computations must be employed in order to take into account the scenario dynamism.

- The not-admissible edges are deleted and the obstacles vertices are included in the tessellation by linking them to the nearest triangles vertices. This approach recovers the obstacles edges but, as the previous approaches, it is not suitable for on-line computations due to the dynamism of the scenario.

It is clear that, using any of the previous techniques, the scenario dynamism is always a problem since the uniform tessellation procedures are not dynamic and hence an heavy on-line computational load is required. For this reason, uniform tessellations are not suitable for real-time purposes. In the remaining of this section, non-uniform tessellation are considered in order to deal with the scenario dynamism.

3.3.2 Constrained Delaunay Triangulation

The Constrained Delaunay Triangulation (CDT) is a modification of the standard Delaunay Triangulation (DT) [15] and it can be applied to a set of points in the plane (as the DT) but with many fixed edges (the constraints). The main property of the DT is that the enclosing circle of each triangle does not contain any point. The CDT has also this property except in the neighborhood of the constraints. Figure 3.3 shows an example of Constrained Delaunay Triangulation.

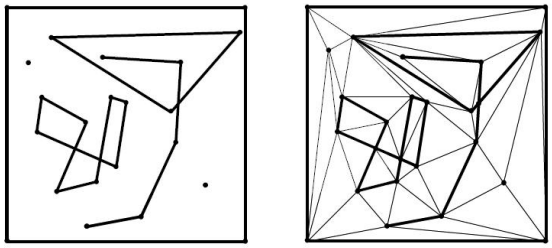

Figure 3.3: Example of CDT. (*left*) non-triangulated scenario; (*right*) triangulated scenario. Note that the constraints are not necessarily closed polygons.

Just for completeness, it is important to notice that the dual *Constrained Voronoi Tessellation* is not suitable for the cooperative control purposes since the resulting tessellating polygons can be overlapped, while the resulting triangles of the CDT are not [15].

The main advantage of using the CDT is that it can be employed into an incremental (and hence dynamic) version; anyway the computational cost of re-building it from the beginning is not high since it is $\mathcal{O}(n \log(n))$ denoting with n the total number of points. More

details about the incremental construction of the CDT can be found in [16].

Kallmann Procedure

In [17, 18] Kallmann proposes a CDT-based approach to the path-planning problem in order to obtain a dynamic procedure. An efficient data structure is employed together with the incremental construction of the CDT [16] to take advantage of the information obtained during the CDT construction. The main steps of the Kallmann procedure are the following:

1. Perform (or update) the CDT
2. Find the two triangles T_i and T_e enclosing the initial and end points using a *random search*
3. Find the shortest chain of adjacent triangles between T_i and T_e (see figure 3.4)
4. Find the shortest path between the two points in the polygon (without holes) defined by the union of the triangles in the chain

The triangles-chain can be found using an *A-search* procedure while the shortest path inside a polygon without holes can be found using a *funneling* algorithm (see figure 3.5). The computational cost of the entire procedure depends mainly on the two last steps because the CDT, as said before, is employed dynamically.

3.3 Suboptimal Algorithms

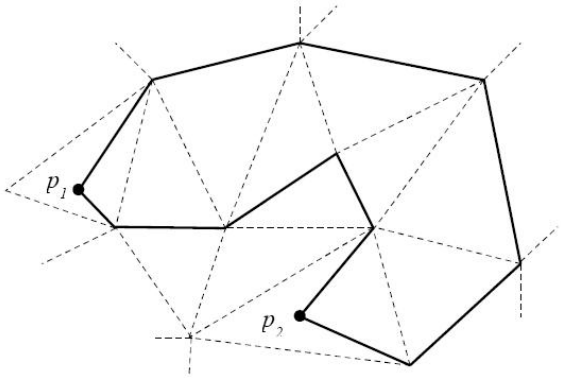

Figure 3.4: Example of triangle chain (solid line) joining two points p_1 and p_2.

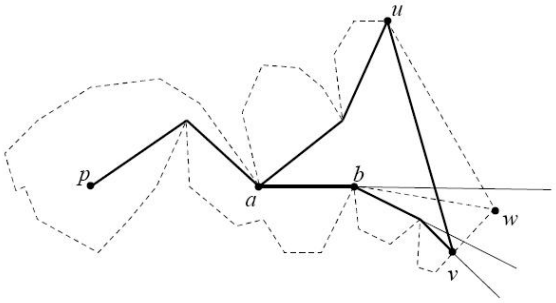

Figure 3.5: Example of *funneling* algorithm.

The resulting path is sub-optimal because the triangles chain is not necessarily the one containing the shortest path between the two points. More details on the present procedure can be found in [18].

3.3.3 Adjacency Path Procedure

This subsection proposes a novel approach to path-planning. The basic step is the incremental version of the CDT, as in the Kallmann procedure but it does not need the *A-search* and the *funneling* algorithm. The resulting path is however not optimal at all and it can be very different from the optimal one, but it has the main advantage of generating an obstacle-free path with a low computational load. After the path is found, a reducing procedure can be employed in order to get a shorter path.

The main point of the proposed procedure is the construction of the *Adjacency Path* ($\mathcal{AP}$) that is the set of segments that link the incenters (the intersection point of the bisectors of the triangle) of the adjacent triangles. The adjacency information can be directly recovered by the DCT and using the efficient data structure proposed by Kallmann, and hence the construction of the $\mathcal{AP}$ is a straightforward and low time-consuming operation.

Adjacency Path The adjacency path is built along the incenters of the adjacent triangles because of the following desirable property:

Proposition 6 *The segment that links the incenters of two adjacent triangles crosses the adjacent edge.*

Proof. In figure 3.6 the two triangles $\overset{\triangle}{ABC}$ and $\overset{\triangle}{DBC}$ are adjacent and the points P_1 and P_2 are their respective incenters. The objective is to proof that the two diagonals of the polygon P_1BP_2C cross inside the polygon; in other words, the objective is to demonstrate that the polygon P_1BP_2C is convex.

The angle β is composed by the sum of the two angles $\widehat{P_1BC}$ and $\widehat{P_2BC}$, and since the internal angles of each triangle are strictly lower than π, then the two considered angles are convex and are such that:

$$\begin{aligned} \widehat{P_1BC} &< \tfrac{\pi}{2} \\ \widehat{P_2BC} &< \tfrac{\pi}{2} \end{aligned} \tag{3.3}$$

Hence the angle β is strictly lower than π. Analogous considerations can be done for the angles $\widehat{P_1CB}$ and $\widehat{P_2CB}$. The polygon P_1BP_2C is composed by the intersection of two convex angles and thus it is convex too. Hence the segment $\overline{P_1P_2}$ is fully inside the polygon, thus demonstrating the proposition.

■

The previous proposition is very useful in finding an obstacle-free path because, once the CDT is built, the triangles enclosed into the obstacles are rejected and if two triangles are adjacent, then both triangles are admissible, hence the adjacent edge is not an obstacle-edge. As a consequence $\mathcal{AP}$ is composed by admissible edges only.

Path Planning Procedure We are now ready to present the entire procedure:

1. Perform (or update) the CDT.

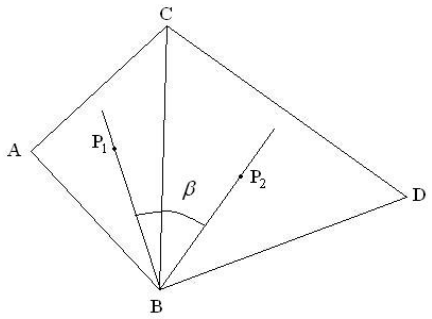

Figure 3.6: Incenters (P_1 and P_2) of two adjacent triangles.

2. Find the two triangles T_i and T_e enclosing the initial and end points.

3. Build (or update) the adjacency path by linking the incenters of the adjacent triangles.

4. Find the shortest path along $\mathcal{AP}$ using the Dijsktra algorithm.

Computational Considerations The CDT is employed incrementally and, with the exception of the first step, it does not affect the computational load of the entire path-planning procedure. Analogous consideration can be made about the search of the two enclosing triangles since, it is reasonable to think that, if the two points move outside their respective triangle, the following enclosing triangle will be one of the (maximum) three adjacent triangles to T_i and T_e thus with a low computational load. The construction of the adjacency path is very fast because it uses the adjacency information

of the CDT. Finally, the Dijkstra algorithm step must be completely re-run at every time step but, since the $\mathcal{AP}$ has a very low number of edges, then the Dijkstra algorithm is very fast.

In conclusion, the total computational load of a single step of the proposed procedure is primarily due to the Dijkstra algorithm only, and hence it is $\mathcal{O}(E + N \log N)$, where $E \leq 3N$ and N is the total number of triangles.

In figure 3.7 an example of the application of the proposed procedure is shown.

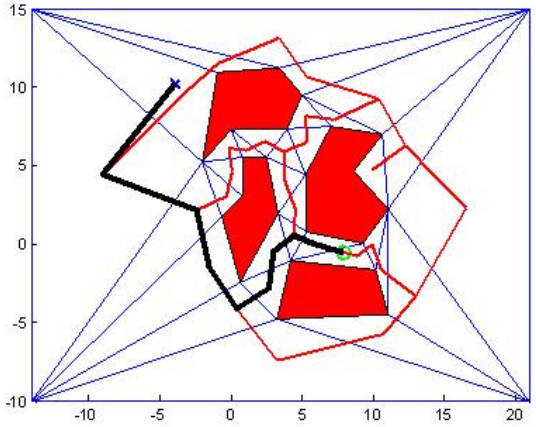

Figure 3.7: Example of path calculated with the *Adjacency Path* procedure. (*black*) obtained path; (*red*) adjacency path.

Path Reduction The result of the proposed procedure is a feasible path and it requires a low computational load. However it can be very different from the optimal path as in the previous example.

In order to shorten the resulting path the following procedure can be employed:

1. Consider three consecutive points $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, $\mathbf{p}_{i+2}$ on the path (excluding the first and the last nodes)
2. Try to directly link $\mathbf{p}_i$ with $\mathbf{p}_{i+2}$ with a straight segment.
3. If the built segment crosses the two adjacent edges of the triangles enclosing the three points then it is admissible and its length is shorter than the length of $\overline{\mathbf{p}_i\mathbf{p}_{i+1}\mathbf{p}_{i+2}}$
4. If the built segment is admissible then delete point $\mathbf{p}_{i+1}$
5. Repeat the entire procedure until no other reductions can be made

The previous procedure is not unique in the sense that, choosing three different initial points the result can be different. However the resulting path is surely shorter than the initial path. An example of the reduction procedure is shown in figures 3.8 and 3.9.

Application The $\mathcal{AP}$-based path planning procedure can be efficiently employed in a cooperative control context since it provides a *safe* path in a very short time. This property allows the use of slower sampling rates or the use of more complex mission management (i. e. task assignment, see chapter 4) algorithms. Moreover, once the *safe* path has been found and the assignment has been performed, the shortest path can be found using an optimal solver like the one presented in section 3.2.

3.3 Suboptimal Algorithms

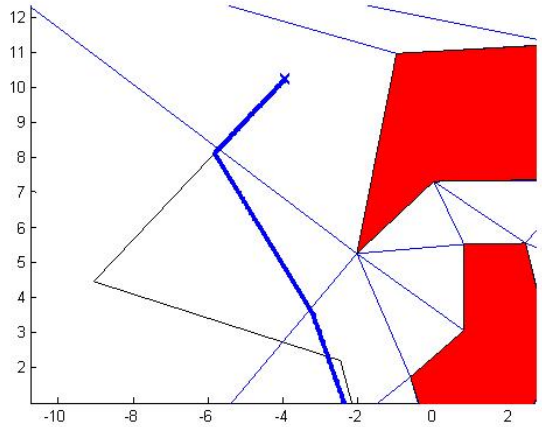

Figure 3.8: Path reduction. (*black*) original path; (*blue*) reduced path.

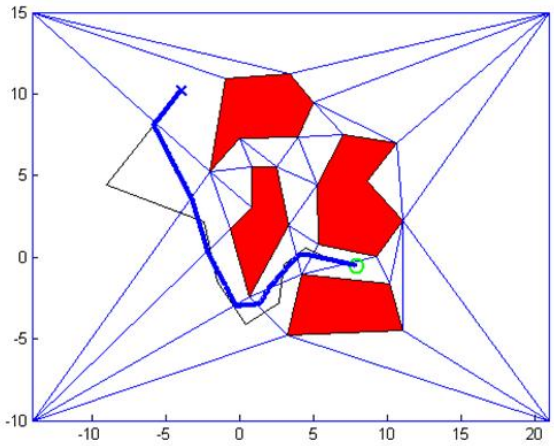

Figure 3.9: Full example of reduced path. (*black* original path; (*blue*) reduced path.

3.4 Concluding Remarks

The problem of path planning in a scenario with many obstacles has been addressed. The optimal solution, that is, the shortest obstacle-free path between two generic points in the plane can be found using some optimal techniques. Though not too computationally expensive, these techniques are not dynamic, and hence are not suitable for application in real (and dynamic) environments. Some approximate but dynamic path-planning techniques are presented. Finally a novel approach based on the incremental construction of the CDT and on the geometric properties of the triangles, has been proposed. Though not optimal at all, this technique is capable of producing an obstacle-free path in a dynamic way. This path can be successively reduce by fast online computations, taking advantages from the geometrical properties of the triangles again. Such technique is then suitable for real-time purposes.

Chapter 4

Task Assignment

In this chapter the problem of assigning the vehicles of a team to the required tasks in the scenario is addressed. As introduced in chapter 1, each task is associated to a target, while a target may have one or more joint tasks. The chapter deals with the most challenging problem of cooperative control because finding the optimal solution often is numerically expensive and not feasible for real-time implementation. For this reason a sub-optimal, but more numerically feasible, solution is sought. The performances of sub-optimal algorithms are compared (where possible) with the optimal solution, found using standard off-line solvers, otherwise compared in terms of residual cost function value (see [23]).

The chapter is organized as follows: in section 4.1 the task-assignment problem is defined. Then a rigorous way of finding the optimal solution is presented. A set of existing sub-optimal approaches to task-assignment problem are reviewed next, together with two novel approaches based on *targets clustering* and *dynamic*

task ranking. Some concluding remarks end the chapter.

4.1 Problem Definition

In chapter 3 the problem of finding the shortest obstacle-free path between two points in the plane was addressed using several different methods. The results of the path-planning algorithm are now used as input data for the task assignment problem.

Consider a generic vehicle v, a generic target t together with two joint tasks τ_1 and τ_2. Both the vehicle and the target are assumed to be dimensionless points ($\mathbf{p}_v$ and $\mathbf{p}_t$) in the plane. Assume that τ_1 must be accomplished before the other task and, for simplicity and without loss of generality, that both tasks are of *visit* type. Finally assume that the cost $c_{v\tau_1}$ that v has to serve τ_1 is given by the length of the previously calculated obstacle-free path between $\mathbf{p}_v$ and $\mathbf{p}_t$. Since both tasks are associated with the same target, then the relative *distance* between them is trivially zero, hence a different and reasonable choice must be done in order to establish a correct cost between τ_1 and τ_2.

The simplest way to deal with this problem is to consider the length of a circumference that a vehicle must follow to come back to a point, once it is passed over it, neglecting for the moment the presence of obstacles. This length clearly depends on the vehicle's characteristics (that is, the *minimum turning radius*). For the purposes of this chapter, the length of such circumference is not relevant and hence the cost between τ_1 and τ_2 is simply set to a positive value. The results obtained here are completely independent of this value.

A similar consideration can be made if the two tasks were associated to two different targets.

4.1.1 Assignment Graph

Consider the set of vehicles V and the set of tasks Υ. Following the previous considerations it is straightforward to define the costs between each pair vehicle-task and task-task. An effective way to represent the input data for the task-assignment is to group the vehicles and tasks into a unique graph (from now referred as the *Assignment Graph*, $\mathcal{AG}$) as follows:

$$\mathcal{AG} = \{\mathcal{N}_{AG}, \mathcal{E}_{AG}\} \tag{4.1}$$

The nodes of $\mathcal{AG}$ ($\mathcal{N}_{AG}$) represent either the vehicles and the tasks, while the edges ($\mathcal{E}_{AG}$) represent the feasible routes between the pairs vehicle/task and task/task. The weight (c_{ij}) of each edge linking two generic nodes i and j, represents its associated cost. Since the objective of the task-assignment is to make the vehicles to serve all the tasks once, then the nodes representing the vehicles must have outgoing edges only, while the other nodes may have both incoming and outgoing edges.

A feasible assignment is a set of routes starting from the vehicles, such that all the tasks are served once only. It is worth noticing that if a task had to be served twice or more, it is possible to split the task into two or more nodes of $\mathcal{AG}$ and then each *task-node* must be visited once only. In order to get a correct definition of a feasible assignment it is necessary to add a fictitious node η_F to $\mathcal{AG}$. This node is such that the weights between η_F and the vehicle nodes are zero while

the weights between η_F and the task nodes are infinity. With the previous statements it is possible to define a *feasible assignment* as a *directed covering tree* of $\mathcal{AG}$ without bifurcations except for the first node η_F. More formally the definition of a feasible assignment can be given as follows:

Definition 7 (Feasible Assignment) *A feasible assignment* $\mathbb{A}$ *for a given assignment graph* $\mathcal{AG}$ *is a directed covering tree, rooted at* η_F*, without ramifications except between* η_F *and the vehicle nodes and without sub-tours:*

$$\mathbb{A} = \{E_{ij} | E_{ij} \in \mathcal{E}_{AG}\} \tag{4.2}$$

such that:

- Each node has at most one incoming edge
- Each node has at most one outgoing edge, except η_F that has exactly N_{veh} outgoing edges.
- The *vehicle-nodes* have not incoming edges from the task nodes
- Each *task-node* has one incoming edge
- There are at most N_{veh} branches starting from a *vehicle-node* each.

The last condition is necessary to avoid sub-tours, that is an isolated set of *task-nodes*. It is easy to prove that the maximum number of branches in the tree must equal to the number of vehicles, otherwise there will be surely a bifurcation in one or more branches.

The cost $C_{\mathbb{A}}$ of a feasible assignment is given by the total sum of the edge weights in $\mathbb{A}$, that is the cost of the *covering tree* and it is given by (4.3):

$$C_{\mathbb{A}} = \sum_{i,j} c_{ij} \qquad \texttt{s.t.} \;\; E_{ij} \in \mathbb{A} \tag{4.3}$$

The general problem of task-assignment can thus be defined as follows:

Problem 8 *Given an assignment graph $\mathcal{AG}$, find the minimum-cost feasible assignment $\mathbb{A}_{MC}$.*

An alternative way to represent $\mathcal{AG}$ is to define two matrices $M_{vt} \in \mathbb{R}^{N_{veh} \times N_{task}}$ and $M_{tt} \in \mathbb{R}^{N_{task} \times N_{task}}$, where the indices vt and tt stand for *vehicle-to-task*, and *task-to-task* respectively. The entries of both matrices are the weight values of the corresponding edge in the $\mathcal{AG}$ and hence, neglecting the fictitious node η_F, there is a one-to-one agreement between the two just defined matrices and $\mathcal{AG}$.

In a similar way, a feasible assignment $\mathbb{A}$ can be expressed in matrix form introducing two joint matrices A_{vt} and A_{tt} of the same dimensions of M_{vt} and M_{tt} respectively. The two matrices A_{vt} and A_{tt} may have zeros or ones entries with the following meaning:

- $A_{vt,ij} = 1$ means that the i^{th} vehicle is assigned to the j^{th} task.

- $A_{tt,ij} = 1$ means that the vehicle (there will be surely one for the posed constraints) that accomplishes the i^{th} task then must accomplish the j^{th} task.

The definition of a feasible assignment then becomes, in matrix form, as follows:

$$\mathbb{A}_M = \{A_{vt}, A_{tt}\}$$

$$\begin{aligned}
\texttt{such that } & \sum_{i=1}^{N_{veh}} \sum_{j=1}^{N_{task}} A_{vt,ij} \geq 1 \\
& \sum_{j=1}^{N_{task}} A_{vt,ij} \leq 1 \quad \forall i = 1 \ldots N_{veh} \\
& \sum_{j=1}^{N_{task}} A_{tt,ij} \leq 1 \quad \forall i = 1 \ldots N_{tar} \\
& \sum_{i=1}^{N_{veh}} A_{vt,ij} + \sum_{i=1}^{N_{task}} A_{tt,ij} = 1 \quad \forall j = 1 \ldots N_{task}
\end{aligned} \tag{4.4}$$

In addition to the previous constraints, there must be checked that there are not sub-tours among the *task-nodes* since this condition will cause two or more tasks not to be accomplished. In the next section a more rigorous way of establishing a feasible assignment is presented, also including the *no-sub-tours* constraints.

Example 9 *A particular case of the previous problem is a scenario with a single vehicle and a set of tasks without precedences. This is the well known* Traveling Salesman Problem (TSP) *and it is known to be a* NP-hard *combinatorial problem. Consider for instance the scenario shown in figure 4.1 where the white circle represents the vehicle and the black circles represent a set of targets; assume in addition that there are not obstacles. In the same figure the optimal route that starts from the vehicle and visits each target is shown.*

4.1 Problem Definition

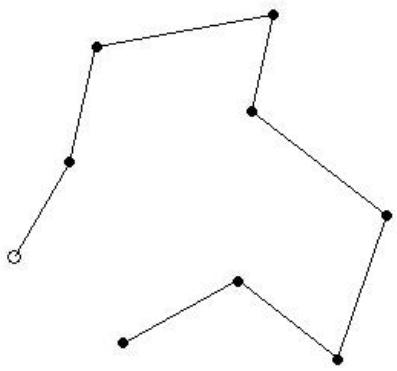

Figure 4.1: Traveling Salesman Problem. (*white circle*) vehicle; (*black circles*) targets.

Assuming in addition that the vehicle has a minimum turning radius, the problem can be solved following for instance [19, 20].

In the considered case, the total number of possible routes is $8! = 40320$ *and since the problem is* NP-hard *then it is necessary to compute all the possible routes and select the cheapest one.*

In the more general case, finding the optimal route is a much harder problem because of the presence of many vehicles, yielding then a larger number of possible routes that is given by $N_{veh}^{N_{task}} N_{task}!$ (see for instance [23]). This is known as the *Multiple Traveling Salesmen Problem* (MTSP). A simple scenario with three vehicles and five tasks yields about 30000 total possible routes and then in more complex scenarios it is not possible to compute all the assignments and choosing the cheapest one since it would take a too large computational time. Alternatively the optimal solution can be found em-

ploying a linear programming problem but the computational cost of such technique is still too high and hence sub-optimal, but faster, procedures are required. In the remaining of the chapter a set of existing solution algorithms are reviewed, and next two novel approaches to solve the task assignment problem are proposed.

4.2 Optimal Solution

Let A be a matrix resulting from the vertical matrix concatenation of A_{vt} and A_{tt}, and denote with a_{ij} the generic entry of A, then the task assignment problem can be formalized into a dynamic programming problem as follows:

$$\begin{aligned}
\min_{\mathbf{a}} J = \min_{\mathbf{a}} & \sum_{i=1}^{N_{veh}+N_{task}} \sum_{j=1}^{N_{task}} c_{ij} a_{ij} \\
\texttt{subject to} \quad & \sum_{i=1}^{N_{veh}+N_{task}} a_{ij} = 1 \qquad \forall j = 1 \dots N_{task} \\
& \sum_{j=1}^{N_{task}} a_{vj} \leq 1 \qquad \forall i = 1 \dots N_{veh} \\
& \sum_{i=1}^{N_{veh}} \sum_{j=1}^{N_{task}} a_{ij} \geq 1 \\
& \sum_{i,j} a_{ij} < \mathbb{N}(S_k) \qquad \forall i,j \ \texttt{s.t.} \ a_{ij} \in S_k
\end{aligned} \tag{4.5}$$

where the optimization variables a_{ij} are binaries. The term $\mathbb{N}(.)$ is the *number-of-nodes* operator and S_k represents a subset of $\mathcal{N}_{AG}$ containing at most $(N_{task} - 1)$ *task-nodes* and no *vehicle-nodes*. This

kind of *linear binary programming* can be solved by dedicated numerical solvers (such as CPLEX and *Mathematica*).

However the number of optimization variables grows very much as the scenario size increases. Consider, as an example, a simple scenario with four vehicles and six tasks: in this case the number of optimization variables is 60. In the general case the number of variables is given by $N_{task}\,(N_{veh} + N_{task})$, and typically the number of tasks is larger than the number of vehicles, and hence the complexity grows with $\mathcal{O}\,(N_{task}^2)$. In addition, this binary programming problem is proven to be *NP-Hard* and hence the worst-case computational load may become unfeasible for real-time applications. Moreover the entire solving procedure is fully centralized and, especially as number of tasks increases, a very powerful computational unit would be required. Such computational unit is not allowable in real-time implementations since the vehicles have limited computational power.

In conclusion, the main disadvantage of exactly solving this kind of problem is the high computational cost. For this reason a suboptimal, but more computationally feasible, solution is sought.

4.3 Suboptimal Solution

This section focuses on the problem of finding a suboptimal solution to the assignment problem 8 in order to achieve satisfactory performances (in terms of resulting assignment cost) taking into account the implementation issues, that is, low computational load, decentralized and dynamic procedures. As Rasmussen *et al.* said in [23]

the only way of establish the degree of optimality of the suboptimal procedures is to compare the results in terms of a given cost index because, in the general case, the optimal solution is not known or it is very hard or too computationally intensive to be found. In this work the optimal solution of all the problems is found formulating the problem as in section 4.2 and using the *LinearProgramming* command of *Mathematica*. The results of the suboptimal task-assignment algorithms are compared with the optimal one. The next subsections present many centralized and decentralized task-assignment procedures focusing on the implementation issues and the computational load.

4.3.1 Hungarian Algorithm

The Hungarian Algorithm (also known as the Munkres' algorithm [25, 26]) is a simple and fast optimization algorithm and is capable of producing an optimal assignment with respect of the following minimization problem:

$$\begin{aligned} \min_{x_{ij}} J &= \min_{x_{ij}} \sum_{i=0}^{N_{veh}} \sum_{j=0}^{N_{task}} c_{ij} x_{ij} \\ \texttt{subject to} \quad & \sum_{i=0}^{N_{veh}} \sum_{j=0}^{N_{task}} x_{ij} = \min\{N_{veh}, N_{tar}\} \\ & \sum_{i=0}^{N_{veh}} x_{ij} \leq 1 \ \ \forall j \\ & \sum_{j=0}^{N_{tar}} x_{ij} \leq 1 \ \ \forall i \end{aligned} \tag{4.6}$$

4.3 Suboptimal Solution

Assuming $N_{veh} = N_{tar}$ the optimal solution produces an assignment in which each vehicle is assigned to a single task and viceversa. If $N_{veh} > N_{tar}$ then there will be unassigned vehicles; otherwise, if $N_{veh} < N_{tar}$ there will be unassigned tasks. The exact solution of problem (4.6) can be found using a standard *linear binary programming*, however, the Hungarian algorithm provides a simpler way to solve this optimization problem. The details of the algorithm can be found in [25] while a parallel and asynchronous version of the algorithm can be found in [27].

Resulting Assignment

The application of the Hungarian Algorithm to the task-assignment problem needs to be refined in order to get a feasible assignment, as the one defined at the beginning of this chapter. In fact the result of the Hungarian Algorithm can not be directly used as a feasible assignment for the following reasons:

- Independently by the relation between N_{veh} and N_{task} the possibility that two or more tasks are *close* each other, and could be served by the same vehicle in sequence is neglected.
- If $N_{veh} < N_{task}$ there are unassigned tasks.

In order to get a feasible (but not necessarily *minimum-cost*) assignment a kind of *simulation* must be employed. The proposed simulation procedure is presented next:

1. Initialize the Hungarian Algorithm using the matrix M_{vt}
2. Run the Hungarian Algorithm on M_{vt}

3. Consider the assigned vehicles and take the one with the lowest cost (also considering the partial accumulated costs) and simulate the fact that it serves its assigned task while the other vehicles remain at their positions.

4. Update the matrix M_{vt} assuming that the selected vehicle has moved to serve its task while the other vehicles have not changed their position; update the partial cost of the selected vehicle and remove the served task. Go to step 2.

The previous iterative procedure ends whenever each task has been visited. The final assignment is obtained by recording the task-visits sequence for all the vehicles. This assignment is feasible for construction though it is not necessarily the optimal one with respect to (4.5).

Implementation Issues

The complexity of the Hungarian Algorithm is $\mathcal{O}\left(N_{veh}^3\right)$ (see for instance [38]) and hence it is less computationally expensive than exactly solving problem (4.5). However the resulting assignment is not optimal at all because the underlying constrained optimization problem (4.6)is different from (4.5). In particular, simulations have shown that the resulting assignment is myopic in the sense that it neglects the *proximity* of the tasks. The procedure explained in the previous paragraph, while producing a feasible assignment, partly reduces this problem because the resulting assignment is neglected except for the vehicle with the lowest cost.

A distributed version of the Munkres algorithm can be found in [39]. It relies on the concurrent implementation of the steps of the

Hungarian algorithm. Obviously this decentralized version is much more interesting because it avoids the presence of a powerful central unit. However, the Hungarian algorithm is not a dynamic at all and then whenever scenario changes the entire procedure must be run again from the beginning.

In conclusion, the Hungarian algorithm is a fast and distributed task assignment procedure but it has the drawback of being myopic and not dynamic at all. For this reason other task-assignment techniques are sought.

4.3.2 Auctioning

The *auctioning* is a decentralized procedure which is capable of producing a sub-optimal assignment among a team of cooperating vehicles. This procedure [40, 41, 28] is based on a dynamic mechanism of prices and bids among the vehicles of the team. Roughly speaking, a vehicle act as an auctioneer and establishes a price of a given task. Then the vehicles of the team (also including the auctioneer) try to get the task by increasing their bids. After a while, the auctioneer announces the winner of the bid and the winning bidder is assigned to that task. The *auctioning* paradigm allows multi-vehicle task assignment by the definition of a required number of vehicles for each task. In the remaining of this subsection more details about the auctioning are reviewed.

Input Data

Auctioning mechanism requires the definition of many variables that are listed below:

$util_i$ The utility (or benefit) of a given task i. It is assumed that all the vehicles have the same utility value for a given task.

req_i The number of required vehicles to serve the task i.

asn_i The number of already assigned vehicles to task i.

$price_i$ The price of task i.

Decision Dynamics

In the *auctioning* procedure each vehicle can act as an auctioneer or a bidder. It is important to focus on the fact that this two behaviors do not exclude each other. In other words, a vehicle can be the auctioneer and a bidder for a given task at the same time. In this paragraph both dynamics are explained.

Auctioneer Dynamics The *auctioneer* has the objective of establish the price of a task. It keeps a decreasing list of the received bids (*received_bids*) coming from the vehicles of the team. Following [28], the *forward* dynamics is given by the following expression:

$$price_t = \begin{cases} received\_bids[req_t] & \text{if} \quad \text{num}(received\_bids \geq req_t) \\ \min received\_bids[i] & \text{otherwise} \end{cases} \tag{4.7}$$

After the end of the round the auctioneer chooses the best bids (depending on the value of *req*) and then tries to turn these provisional commitments into final ones by negotiating with the winning bidders. This is a necessary step because each bidder may send a *bid-retract-request* signal reflecting the fact that it has changed its best task. If

the auctioneer fails to turn commitments into final ones, then the *reverse step* is performed by reducing the price of a $price_reduce_ratio$, otherwise the winners are announced and the auction ends.

Bidder Dynamics Each bidder makes an offer for its best task. This offer depends on the set of task prices of the active auctions. The offers of all the bidders are collected by the respective auctioneers and whenever a bidder changes its best task, it sends a *bid-retract-request* signal to the respective auctioneer. The decision dynamics of the bidder are the following:

- Calculate the benefit a_{ij} of servicing task j

$$a_{ij} = \frac{util_j}{req_j - asn_j} - c_{ij} - price_j \tag{4.8}$$

- Find j_{i_1} and j_{i_2} as follows:

$$\begin{aligned} j_{i_1} &= \arg\max\left(a_{ij}\right) \\ j_{i_2} &= \underset{j \in \text{tasks} - \{j_{i_1}\}}{\arg\max}\left(a_{ij}\right) \end{aligned} \tag{4.9}$$

- Bid for task j_{i_1} with value $\pi_{j_{i_1}} = a_{ij_{i_1}} - a_{ij_{i_2}}$

Dynamic Maintenance of Auction Results The auction results are near-optimal at the time of calculation [41]; however this optimality degree can be lost during the mission because of the scenario changes. Since running again the auctions can be very time consuming, then a dynamic mechanism must be employed. in [28] Ahmed *et al.* propose a swapping technique in order to recover the

optimality as mission evolves without running the auction from the beginning. This mechanism requires are large amount of data communication among the team and for this reason it is necessary to establish a threshold value within do not perform any swap (see the referenced paper for the details).

Implementation Issues

The auctioning mechanism requires that the vehicles have a particular internal architecture especially for the management of the auctions. In addition it is necessary to establish which one of the vehicles must act as the auctioneer. The approach proposed in the reference paper is to mark as the auctioneer the vehicle which is the nearest to a given target. Clearly, following this method it could be possible to have a single vehicle that act as the auctioneer for all the tasks thus resulting in a central point for the rest of the team. Once a new task has been recognized the auction can be passed to another vehicle thus avoiding the previous problem; however this cause a larger amount of data exchange among the team.

Simulations have shown that the resulting assignment has the same *myopic* properties of the Hungarian algorithm, though it is a dynamic, and hence preferable, procedure.

4.3.3 Targets Clustering

The objective of this section is to provide a systematic way of reducing the myopic view of the scenario of the auctioning and the Hungarian algorithm. Along this section we talk about target instead of task, however the results can be directly applied substituting the

term *target* with *task*. Observations have shown that both algorithms do not consider any target proximity, that is, it is possible that two *close* targets are assigned to different vehicles. In order to better allocate the resources (the vehicles), *close* targets can be grouped into many clusters reflecting the fact that if a vehicle must visit one target of a cluster, then it will probably visit the other targets.

Consider a simple obstacle-free scenario composed by targets only (the presence of the vehicles is unnecessary for the purposes of the clustering); each target is characterized by its xy-coordinates in the plane. There are not assumption on the targets configuration in the plane and the relative distance between two targets is given by the length of the segment that links the two corresponding points. The considered points can be grouped into many clusters following for instance a *k-means* or a *c-means* algorithm but there is the need of a predetermined number of clusters. Since in a cooperative control context there can be several different situations, it is preferable to employ a procedure which is capable of automatically find the right number of clusters. For instance one may follow Rosenberger procedure in [42], where a set of points are grouped with respect to their relative Euclidean distance.

For what we have said in previous sections, it is important to have a dynamic procedure in order to do not run the entire procedure from the beginning. In addition dynamic scenario events (such as, for instance, target recognition and target destruction) must be treated. Hence a dynamic number of target assessment is sought. To this end the following dynamic procedure is proposed:

1. Initialize the procedure with one cluster only and evaluate its

centroid.

2. Check if two or more centroids are too close (w.r.t. a predefined *proximity* value) and in this case, merge these into a single centroid and decrease accordingly the total number of clusters.

3. Perform a *c-means* algorithm using the current number of clusters.

4. Check if the clusters' size are smaller than a fixed parameter ρ. In this case, stop; otherwise, consider the cluster with the maximum size and add a new centroid placed at the target that is at the maximum distance from its centroid, and go to step 2.

The flux diagram of the proposed clustering algorithm is shown in figure 4.2.

Each vehicle can perform its own clustering and then in order to get a common result among the team, it is assumed that each vehicle has the same information of the other vehicles. This is not a strong hypothesis since, whenever a vehicle recognizes a target it can communicate its position to the others and the target position becomes a team knowledge. For this reason we can study the algorithm as it were centralized, holding in mind that it will be applied autonomously by each vehicle.

The proposed procedure is capable to deal with scenario with many obstacles introducing a modification in the relative distance function. Since the presence of the obstacles may prevent the straight line between two generic points, it is necessary to establish another

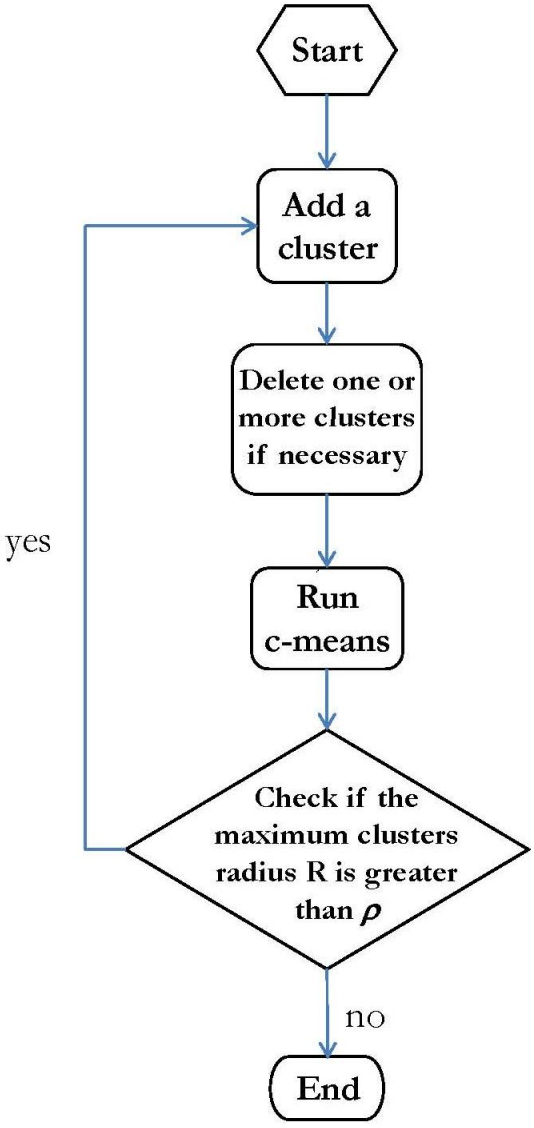

Figure 4.2: Dynamic clustering algorithm flux diagram.

distance function between the targets. The simplest way is to consider the length of the shortest obstacle-free path between each pair of targets. The clustering algorithm is then run over a N_{tar}-dimensional space: each target is associated to a N_{tar}-dimensional vector whose entries contain the relative distances between the given target and the other targets. We can concatenate all these vector into a unique $N_{tar} \times N_{tar}$ matrix D with the following properties:

- $D_{ii} = 0$ for all $i = 1, \ldots, N_{tar}$
- D_{ij} is the relative distance between target i and target j
- The matrix D is symmetric
- $D_{ij} \geq 0$ for all $i, j = 1, \ldots N_{tar}$

The resulting centroids are N_{tar}-dimensional points into a $\Re^{N_{tar}}$ space and the distance between these centroids and the target-points can be evaluated using a generic norm.

Effects of ρ

The value of ρ in the proposed algorithm can be used as a tuning parameter that can be modified during the mission in order to take into account various situations. For instance, consider a team of vehicles that is approaching the mission field. In this case ρ can be set to a large value resulting then into a unique cluster. This is a reasonable choice since the objective of the team is to get close to the action field. Whenever the vehicles are closer to the targets, the value of ρ can be set to a smaller value in order to perform a

finer clustering and then specialize the mission objectives. The lower bound of ρ is zero, yielding then a number of clusters equal to N_{tar} (see figure 4.4 where a small value of ρ is considered).

The capabilities of the algorithm for different values of ρ are shown in the following example.

Example 10 *Consider the scenario shown in figure 4.3 (the vehicles are omitted). The clustering algorithm is performed using different values of the parameter ρ. The results are presented in figures 4.4, 4.5, 4.6. The numbers over the target points represent the cluster identification number which the target belongs to.*

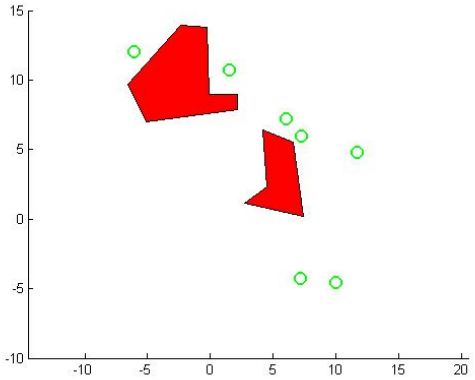

Figure 4.3: Sample scenario. (*circles*) targets.

As expected, increasing values of ρ produce a lower number of clusters containing a larger number of targets.

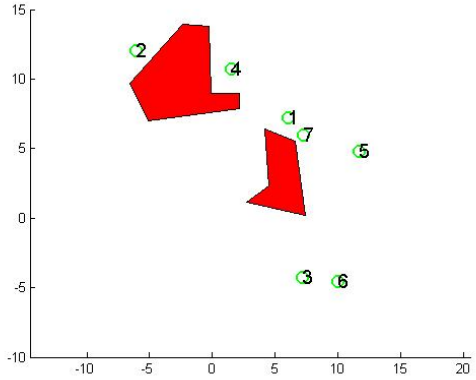

Figure 4.4: Clustering with $\rho = 1$.

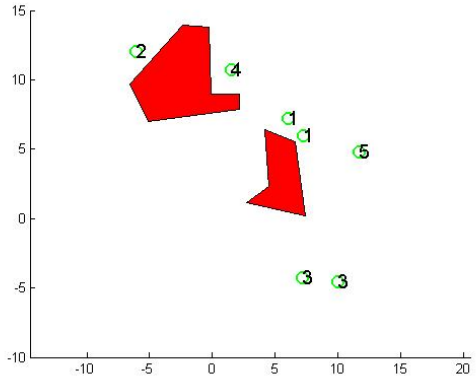

Figure 4.5: Clustering with $\rho = 3$.

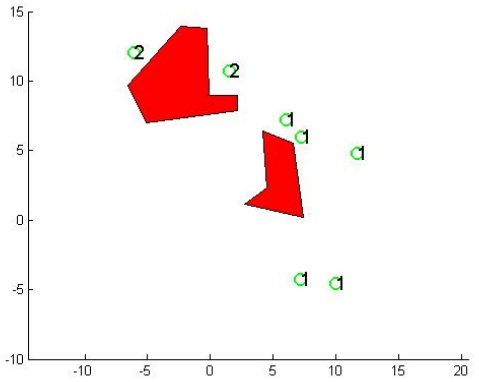

Figure 4.6: Clustering with $\rho = 15$.

Cluster Assignment

Once the targets (or the tasks) are grouped into one or many clusters, the vehicles have to establish their cost to serve each cluster instead of each task. A way to calculate these values is to consider, for each vehicle, the mean cost of the tasks into each cluster. These costs can be rearranged into a $N_{veh} \times N_{clust}$ matrix whose entries represent the relative costs between each vehicle and the clusters.

Assuming that the clustering is performed in a centralized manner, both a centralized and a decentralized task-assignment procedure can be employed. Assuming that each vehicle has the same knowledge of the clusters, it is possible the employ a decentralized procedure to make the *cluster*-assignment.

The results of the cluster assignment is much less myopic with respect to the simple application of a task-assignment procedure

without clustering, depending on the value of ρ. If ρ tends to zero the results are the same as if the clustering was not performed, since $N_{clust} = N_{tar}$. Many examples of the proposed algorithm are presented in chapter 5.

4.3.4 Dynamic Task Ranking

In this subsection a novel approach to decentralized cooperative task assignment is proposed. The main idea is to allow the vehicles to autonomously establish a ranking of the tasks such that each vehicle has its best task depending on the rest of the team choices. The decision dynamics are based on the common human behavior when a team of people are discussing who have to accomplish a given set of tasks: everyone has its own benefits for each task and, depending on the benefits of the other people he may increase or decrease its benefits. These dynamics are formalized into a nonlinear dynamic system built as the concatenation of N_{veh} smaller decision dynamics systems. In the following the proposed approach is presented together with many properties.

Benefits Definition

Define a benefit b_i of a vehicle to serve a given task i as in (4.10):

$$b_i = \frac{1}{c_i} \tag{4.10}$$

where c_i is the the cost of the task. Denote then with b_{ij} the benefit of servicing task j after the completion of task i in an similar way.

4.3 Suboptimal Solution

The resulting total benefit B_i of a given task i, for a given vehicle, is given by the sum of the partial benefits as in (4.11).

$$B_i = b_i + \sum_{j=1}^{m} b_{ij} \tag{4.11}$$

It follows that a task has a larger benefit if it is *close* to other tasks, that is, if the value of b_{ij} is high, then if a vehicle serves task i then it will probably serve task j. In this view the previous definition of the benefits implicitly includes a kind of tasks clustering.

An important consideration of the assessment of the values of b_{ij} is that each vehicle can estimate its own benefit independently by the others and hence the vehicle's capabilities can be directly considered in the benefit definition. For instance one or more tasks could not be accomplished by a vehicle, and then it traduces into an infinite cost and hence an associated null benefit. For the same reason, vehicles capabilities can affect the relative *task-to-task* benefit value.

Decision Dynamics Definition

Once a vehicle has autonomously defined the benefit of each task, the tasks are sorted following a decision dynamics depending on some internal variables (called *weights*) and some external values coming from the other vehicles. The internal state-variables of the dynamics represent the weights that each task has for the vehicle. The generic weight w_{vt} is associated to the vehicle v and to the task t, and its dynamics are expressed in (4.12)

$$\dot{w}_{vt} = (1 - \lambda_v) \left(1 - \sum_{j=1}^{N_{task}} w_{vj} \right) B_{vt} - \lambda_v \left(\sum_{i=1}^{N_{veh}} a_{iv} B_{it} w_{it} - B_{vt} w_{vt} \right) w_{vt} \tag{4.12}$$

where λ_v is called the *cooperation parameter* of vehicle v and denotes its cooperative or selfish behavior; a_{ij} are binary variables and represent the communication between the vehicle i and the vehicle j, if $a_{ij} = 0$ then there is not communication between the two vehicles. The dynamics expressed in (4.12) can be split into two parts: the former can be viewed as the *non-cooperative* dynamics, while the latter (the one multiplied by λ_v) is the *cooperative* part. The employed convex combination of the two parts allows to establish that each λ_v is in the interval $[0, 1]$, reflecting a fully selfish ($\lambda_v = 0$) or fully cooperative ($\lambda_v = 1$) vehicle. If $\lambda_v = 0.5$ then the corresponding vehicle v is said to be *neutral*, which means that its behavior is equally selfish and cooperative.

The cooperative part tends to decrease the value of w_{vt} while the other part tends to increase its value. This is a behavior reflecting the fact that if a vehicle v discovers that many other vehicles have great benefits to serve a task t, then that task will be probably visited by one (or more) of those vehicles and then, in the view of a team objective, a good choice for v is to try to serve another task by decreasing its weight on t.

With the exception of the extremum values of λ_v, the global dynamics are then the result of a tradeoff between cooperative and non-cooperative behavior of the vehicles of the team.

4.3 Suboptimal Solution

Decision Dynamics Properties The proposed dynamics are now analyzed in order to establish many useful properties. First of all, the weights derivatives are rearranged into a single ODE system:

- Denote with $f_{vt}(W)$ the right-hand term of (4.12)
- $W(t) = \bigcup_{\substack{i=1,\dots,N_{veh} \\ j=1,\dots,N_{task}}} w_{ij}(t)$
- $w_{ij}(0) \in \Omega$
- $\Omega = \left\{ w_{vt} \in (0,1) \subset \Re | \sum_{j=1}^{N_{task}} w_{vj} \leq 1 \quad \forall v \right\}$

The time evolution of the weights can be expressed then as the solution of the following standard Cauchy problem:

$$\begin{cases} \dot{W} = f(W) \\ W(0) = W_0 \end{cases} \tag{4.13}$$

With the previous statement the following theorem holds:

Theorem 11 (Well-Posedness) *The Cauchy problem (4.13) admits a solution and this solution is unique.*

Proof. The existence of the solution is a assured by the *Peano's Theorem* while the uniqueness descends from the *Cauchy-Lipschitz Theorem*:

let $f_{vt} = \dot{w}_{vt}$ be the generic weight derivative, then the generic entry of the Jacobian matrix is given by the following relations:

$$\frac{\partial f_{vt}}{\partial w_{ij}} = \begin{cases} -(1-\lambda_v) B_{it} & \text{if } i = v, j \neq t \\ -(1-\lambda_v) B_{it} - \lambda_v s_{vt} & \text{if } i = v, j = t \\ -\lambda_v a_{iv} w_{vt} B_{it} & \text{if } i \neq v \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

where:

$$s_{vt} = \sum_{i=1, i\neq v}^{N_{veh}} B_{it} w_{it} \tag{4.15}$$

From (4.14) and since the B_{ij}, w_{ij} and λ_v are bounded, then the Jacobian matrix has bounded norm. This is a sufficient condition (together with the continuity of $f(W)$) to the Cauchy-Lipschitz theorem to hold, and then for any given initial point W_0 the solution of (4.13) exists and it is unique. As a consequence, the stated problem is *well-posed*. ■

Another important property of the decision dynamics is expressed in the following proposition:

Proposition 12 *Since the initial values W_0 are in the domain Ω then the time evolution of the weights is fully inside Ω*

Proof. The proof of the previous proposition is made for exhaustion: consider first the case $w_{vt} \to 0$ for any v and t, then the cooperative part of (4.12) tends to zero. Hence the only effective part is the selfish one and it is positive. The result is that $\dot{w}_{vt} \geq 0$.

In a similar way: whenever $\sum_{i=1}^{N_{veh}} w_{it} \to 1$ the only effective part for the weights w_{it} is the cooperative one, and it is negative. it follows that $w_{it} \leq 0$ for all $i = 1, \ldots, N_{veh}$.

The previous two considerations prove the proposition. ■

The well-posedness of the problem assures that the global system is at least simply stable, but it does not say anything about the asymptotic stability nor of the presence of stable equilibrium points. Numerical simulations, starting from random admissible initial values, have shown that for a given set of benefits, the equilibrium point is unique and it is asymptotically stable. The same equilibrium point has been found by the numerical solution of the homogeneous nonlinear system associated to (4.13) reinforcing the convincement that the equilibrium point is unique and asymptotically stable.

Assuming that there are at least as tasks as the vehicles a desirable property would be that the best task for each vehicle be different from the other vehicles. In fact, if this property holds, it means that the team self organizes assigning different tasks to different vehicles. However this nice property does not hold in general. Though a formal proof has not been discovered yet, it seems that whenever the static benefits B_i of the tasks are ordered in the same way for each vehicle, then the vehicles all have the same best task.

Implementation Issues In order to apply the *dynamic task ranking* into a real-time system, it is necessary to discretize the dynamics in (4.12). The simplest way to do that is to employ the Euler formula and obtaining the discrete version of the weights derivatives. However this approach leads to approximation errors that may prevent the stability property of the continuous dynamics. A better choice is to employ the exact discretization in the intersampling time. This kind of discretization is directly employable noticing that for each vehicle (that is, the $B_{it}w_{it}$ products in (4.12)), during the intersam-

pling time, the values coming from the other vehicles are constant, thus resulting in an intersampling dynamics as follows:

$$\dot{w}_{vt} = (1-\lambda_v)\left(1-\sum_{j=1}^{N_{task}} w_{vj}\right) B_{vt} - \lambda_v \left(\sum_{\substack{i=1 \\ i\neq v}}^{N_{veh}} a_{iv} B_{it} w_{it}(k)\right) w_{vt} \tag{4.16}$$

where the values $w_{it}(k)$ are constant since the vehicle v has not knowledge about the variation of such variables. The dynamics (4.16) is linear in the variables w_{vt} and hence the value of these variables after a given time T_S is exactly known employing matrix exponentials (see [43] for a detailed description of a robust implementation). Combining all the weights into a unique system, the intersampling dynamics can be written as follows:

$$\dot{W} = A_c W + B_c \tag{4.17}$$

where:

$$\begin{aligned} A_c = -(1-\lambda_v)\mathrm{diag}(\ B_{v,1} \quad B_{v,2} \quad \dots \quad B_{v,N_{task}}\ )\mathbf{1} \\ -\lambda_v \mathrm{diag}(\ s_{v,1} \quad s_{v,2} \quad \dots s_{v,N_{task}}\ ) \end{aligned} \tag{4.18}$$

$$B_c = \left[\ B_{v,1} \quad B_{v,2} \quad \dots \quad B_{v,N_{task}}\ \right]^T \tag{4.19}$$

Since we are interested at the values of $W(t)$ after a sample time T_s the following discrete system is obtained:

$$W(k+1) = A_d W(k) + B_d \tag{4.20}$$

where

$$A_d = e^{A_c T_S} \tag{4.21}$$

$$B_d = \int_0^{T_S} \left(e^{A_c T_S} B_c\right) dt = (A_c)^{-1} \left(e^{A_c T_S} - I\right) B_c \tag{4.22}$$

Varying Benefits Until now it is assumed that the values of the benefits B were static. However, as mission evolves, these values change, thus affecting the stability properties presented above. Assuming that the values of B hold positive at every time, then the simple stability property holds as well. In addition, since each vehicle serves its best task, the weight that the vehicle has to that task increases with time, while the other weights tend to decrease. Whenever a vehicle finishes to serve its best task, then the other weights gain increasing value depending on the other tasks configuration.

The benefits variation establishes a constraint to the expected performance since, in order to obtain a correct assignment it is necessary to wait until the weights have reached the steady state. A way to establish such constraint is to calculate the linearized system in the neighborhood of the equilibrium point and evaluate its settling time T_{SET}. Then, assuming that the initial configuration of the weights is not so far away, after a three times T_{SET} we can assume that the weights have reached their steady values.

Resulting Assignment Since the desired *separation* property does not hold, it is necessary to employ an algorithm which produces a feasible assignment. An effective way is to follow the same procedure

as in 4.3.1 considering the maximum benefit instead of the minimum cost.

Simulations have shown that the results of the dynamic task ranking based assignment tend to minimize the mission completion time thus resulting into a better resource distribution: assume that each vehicle has to move (and then consuming fuel) until the mission has accomplished, then minimizing the mission completion time results in a minimization of the total fuel consumption considering also the fact the vehicles can perform an *empty* task. In chapter 5 many examples are presented comparing the different task assignment procedures in terms of different cost indices.

Hardware Limitations

Consider the discrete dynamics (4.20) and assume that the vehicles are synchronized, that is, each vehicle has a sample time T_S and the sampling instant is the same for all the vehicles. The only data that must be sent through the team are the Bw products in the *cooperative* part of (4.12), and then the total amount of sent *double*s is $N_{veh}N_{task}$. In addition consider a standard data link baudrate of 9600 bit/s, since a *double* is composed by 8 bytes and, in order to send a *double* two additional bytes must be sent too, it descends that the maximum number of transmittable *double*s is given by:

$$D_R = \frac{BR}{10 \times 8} = \frac{9600}{10 \times 8} = 120 \frac{doubles}{s} \tag{4.23}$$

Then, during a sample time T_S the maximum transmittable number of *double*s is $T_S D_R$. As a consequence, each vehicle has a limited time range T_T for transmit its data, and since its has to transmit

exactly N_{task} *double*s the following constraint must hold:

$$T_T \geq \frac{T_S(N_{task}+1)}{D_R} \tag{4.24}$$

Finally, since each vehicle must transmit within a sample time the following relation must hold:

$$\left\lfloor \frac{T_S}{T_T} \right\rfloor \geq N_{veh} \tag{4.25}$$

The previous considerations allows to establish the maximum allowable scenario complexity for a fixed sample time, or, equivalently, a lower bound on the sample time for a given scenario complexity.

Taking $T_S = 1$, for instance, some admissible values of the maximum allowable number of vehicles and number of tasks is shown in table 4.1.

T_S	T_T	N_{veh}	N_{task}
1	.1	10	10
1	0.05	20	5
1	0.025	40	2
1	0.2	5	20

Table 4.1: Relation between T_S, N_{veh} and N_{task}

4.4 Concluding Remarks

The task-assignment problem is perhaps the most challenging problem of cooperative control. The optimal solution (with respect to a

given cost index) is hard to be found since the computational time can be too large. Hence sub-optimal solutions are sought. In addition a fully decentralized procedure is required in order to allow a team to self organize without the need of a central unit. The Hungarian algorithm and the Auctioning method are fast and sub-optimal assignment procedures with the drawback of a myopic view of the scenario. This disadvantage can be reduced employing a task (or target) clustering based on notion of *relative distance* between the tasks (or the targets). To this end a dynamic clustering procedure has been proposed. Finally a novel and fully decentralized task assignment procedure has been proposed and it is based on a *dynamic task ranking*. This procedure is capable of dealing with the dynamism of the scenario and it allows to set an upper bound to the admissible number of vehicles and number of tasks in the scenario, depending on the hardware communication limitations.

Chapter 5

Examples

The objective of the chapter is to present many examples of the problems issued in this work. The following examples are divided into many paragraphs reflecting the thesis structure and are sorted into an increasing scenario complexity order. The presented scenarios are solved using the procedures presented in the previous chapters and the results are compared in terms of the respective cost indexes.

5.1 Path Planning

In this section the comparison between the optimal and the suboptimal path-planning procedures is presented. Since the main objective is to compare the produced trajectories, all the examples contain one vehicle and one target only. The results are compared in terms of length of the calculated path. The examples contain a scenario with many obstacles and are presented into an increasing complexity order. The vehicle (the *start-point*) is shown as a cross while the target

(the *end-point*) is a circle.

Example 13 (One obstacle) *In this example a simple scenario with one obstacle only is presented. The presence of the obstacle prevents the direct link between the vehicle and the target using a straight line. From figures 5.1 and 5.2 it is clear that the CDT-based trajectory is much longer than the optimal one but, the path reduction algorithm is capable of sensibly reduce the path as in figure 5.3. The results of the path planning algorithms are summarized in table 5.1.*

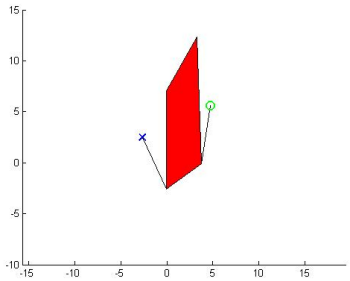

Figure 5.1: Shortest path (1 obstacle)

	Path Length
Optimal	16.0103
CDT - Adj. Graph	31.2498
CDT - Adj. Reduced	24.6250

Table 5.1: Trajectory length comparison(1 obstacle)

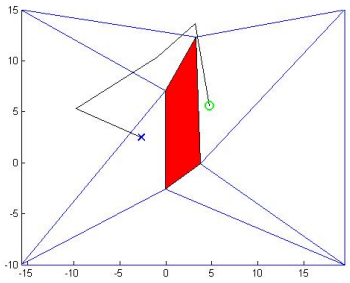

Figure 5.2: Path found using *CDT* with adjacency path (1 obstacle)

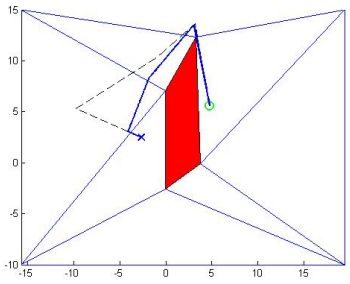

Figure 5.3: Path found using *CDT* with adjacency path and reduced on-line (1 obstacle)

Example 14 (Two obstacles) *This is the same scenario of the previous example where a new obstacle has been added. This obstacle prevents the previous shortest path to be used and hence the new shortest path is (slightly) different from the previous case. Figures 5.4 and 5.5 show the graphic results of the found trajectories. The length of the computed paths are presented in table 5.2. Notice the effectiveness of the online path reduction shown in figure 5.6.*

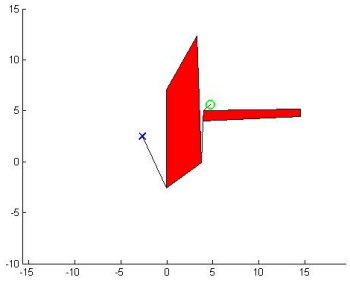

Figure 5.4: Shortest path (2 obstacles)

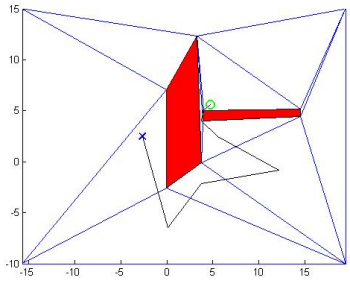

Figure 5.5: Path found using *CDT* with adjacency path (2 obstacles)

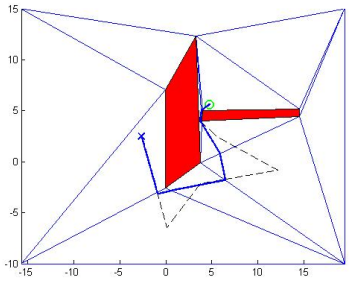

Figure 5.6: Path found using *CDT* with adjacency path and reduced on-line (2 obstacles)

	Path Length
Optimal	16.3242
CDT - Adj. Graph	35.2772
CDT - Adj. Reduced	24.6250

Table 5.2: Trajectory length comparison(2 obstacles)

Example 15 (Three obstacles) *As for the previous example, a new obstacle is added, thus resulting in a scenario with three obstacles. The optimal trajectory is sensibly different from the previous cases since the presence of such concave obstacle makes the* upper *part of the scenario shorter than the* lower *(see figure 5.7). In this case , as in the previous, the CDT-based path planning produces a sensibly different path from the optimal one. However, online path reduction is capable of drastically reduce the computed path length (see figure 5.9.*

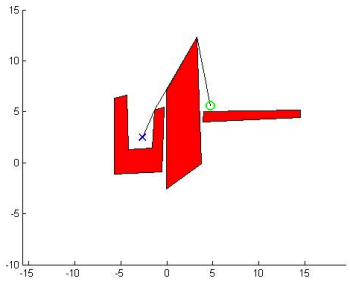

Figure 5.7: Shortest path (3 obstacles)

	Path Length
Optimal	18.3501
CDT - Adj. Graph	38.9851
CDT - Adj. Reduced	24.3750

Table 5.3: Trajectory length comparison(3 obstacles)

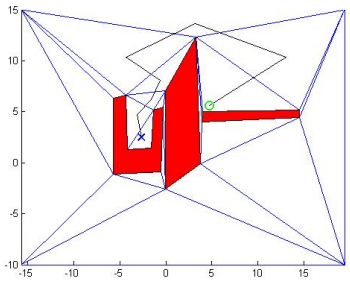

Figure 5.8: Path found using *CDT* with adjacency path (3 obstacles)

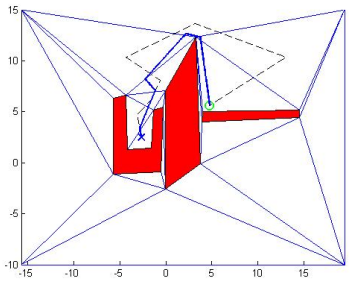

Figure 5.9: Path found using *CDT* with adjacency path and reduced on-line (3 obstacles)

5.2 Task Assignment

The different task assignment procedures are compared with respect to the same cost index that is the team total fuel consumption J_{fuel} (as in (4.5)). In addition the mission completion time J_{time} is presented for each procedure in order to show the differences between the obtained assignments. This index is defined as follows:

$$J_{time} = \max_{v} \{\mu_v\} \tag{5.1}$$

where μ_v is the time that vehicle v takes to accomplish all its tasks. For simplicity the following assumption are made:

- The vehicles have all the same velocity
- The fuel consumption to go from a point to another is equal to the path length.
- A task cost is represented by the length of the path to reach that task
- The completion time of a path is identified by its length (which means that the velocity is assumed to be unitary)

All the examples presented are numerically solved using the *LinearProgramming* command of *Mathematica* obtaining the optimal solution and hence the comparison between the sub-optimal procedures can be made with respect to the optimal cost. In addition, the shortest path between two points has been evaluated using the *visibility graph* approach [1].

[1] In this section the figures contain the visibility graph depicted as light-gray lines

In addition the so-called *fuel distribution* J_{dist} parameter is presented in order to obtain an estimate of the real fuel consumption, especially in the case of aerial vehicles that can not keep still. Using the previous simplificative assumptions, this parameter is defined as follows:

$$J_{dist} = \frac{J_{fuel}}{N_{veh} J_{time}} \tag{5.2}$$

and since

$$J_{time} = \max_{v} \{J_{fuel,v}\} \tag{5.3}$$

and

$$J_{time} \leq J_{fuel} = \sum_{v=1}^{N_{veh}} J_{fuel,v} \leq N_{veh} J_{time} \tag{5.4}$$

it follows that $J_{dist} \leq 1$. As a consequence, the following relation holds:

$$\frac{1}{N_{veh}} \leq J_{dist} \leq 1 \tag{5.5}$$

The optimum value of J_{dist} is obtained whenever each vehicle consumes the same fuel amount, thus resulting in $J_{dist} = 1$. The worst case, instead, is obtained whenever the total fuel consumption (as in (4.5)) is due to a single vehicle that yields $J_{dist} = 1/N_{veh}$.

Example 16 (2 vehicles, 3 targets) *The considered scenario is shown in figure 5.10 and it contains two vehicles and three targets. The results of the procedures explained in chapter 4 are presented in table 5.4.*

Notice that, in the case of the Optimal *and the* Hungarian algorithm *solutions, vehicle* V_1 *accomplishes all the tasks while* V_2 *remains unassigned. Using the other assignment techniques both vehicles are assigned instead.*

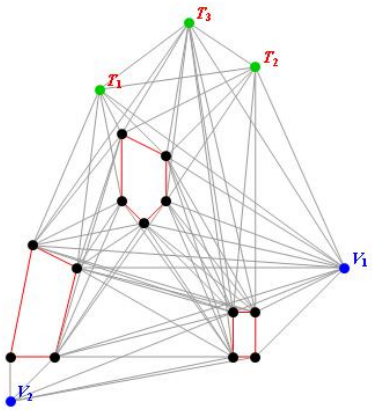

Figure 5.10: Scenario geometry (2 vehicles, 3 targets)

	J_{time}	J_{fuel}	J_{fuel}-err	J_{dist}
Int. Prog.	18.4544	18.4544	0. %	0.5
Hung.	18.4544	18.4544	0. %	0.5
Auct.	14.7148	28.1692	52.6421 %	0.957173
Clust. Hung.	14.7148	28.1692	52.6421 %	0.957173
DTR	14.7148	28.1692	52.6421 %	0.957173

Table 5.4: Scenario results (2 vehicles, 3 targets)

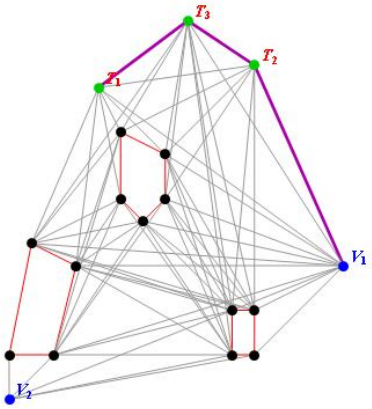

Figure 5.11: Optimal solution and Hungarian algorithm solution (2 vehicles, 3 targets)

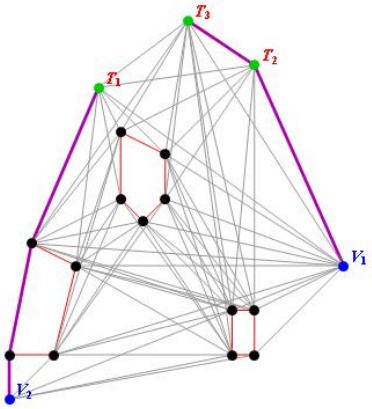

Figure 5.12: Solution using auctioning, clustering and DTR (2 vehicles, 3 targets)

Example 17 (3 vehicles, 5 targets) *The present scenario contains three vehicles and five targets. The scheme is shown in figure 5.13. The results of the application of the procedures explained in chapter 4 are presented in Table 5.5.*

As in the previous example, the DTR *produces a result where all the vehicles are assigned, while the optimal procedure does not. The solution of the* Hungarian algorithm *is the same of the optimal procedure; while the results of the* Clustering *approach are the same of the* DTR*.*

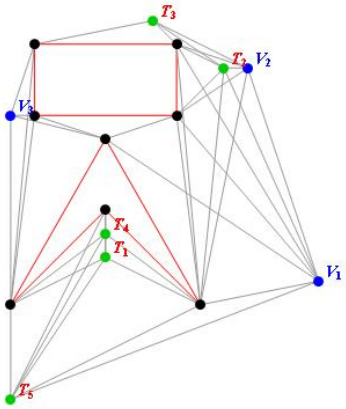

Figure 5.13: Scenario geometry (3 vehicles, 5 targets)

Example 18 (4 vehicles, 7 targets) *Consider the scenario in figure 5.16 with four vehicles and seven targets. The results of the application of the task-assignment procedures explained in chapter 4 are presented in Table 5.6.*

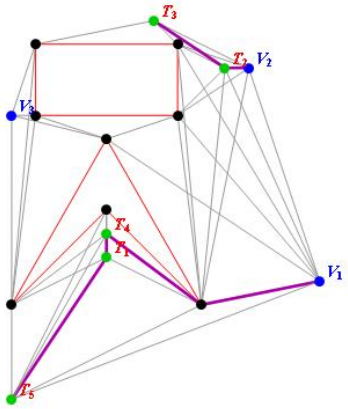

Figure 5.14: Optimal solution (3 vehicles, 5 targets)

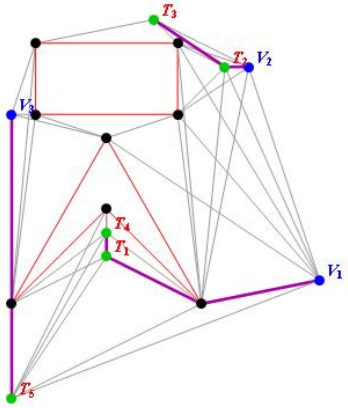

Figure 5.15: DTR solution (3 vehicles, 5 targets)

	J_{time}	J_{fuel}	J_{fuel}-err	J_{dist}
Int. Prog.	18.3101	22.9157	0. %	0.417177
Hung.	18.3101	22.9157	0. %	0.417177
Auct.	27.9	39.4712	72.2454 %	0.471579
Clust. Hung.	12	27.1767	18.5944 %	0.754909
DTR	12	27.1767	18.5944 %	0.754909

Table 5.5: Scenario results (3 vehicles, 5 targets)

Notice, in this case, the high value of the index J_{dist} of the Clustering *and the* DTR.

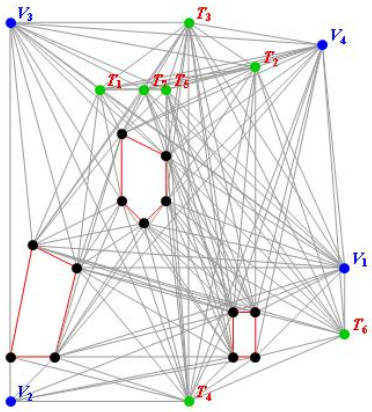

Figure 5.16: Scenario geometry (4 vehicles, 7 targets)

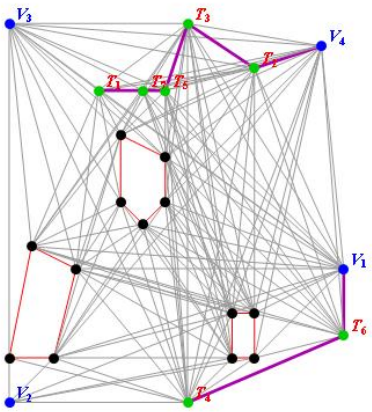

Figure 5.17: Optimal solution (4 vehicles, 7 targets)

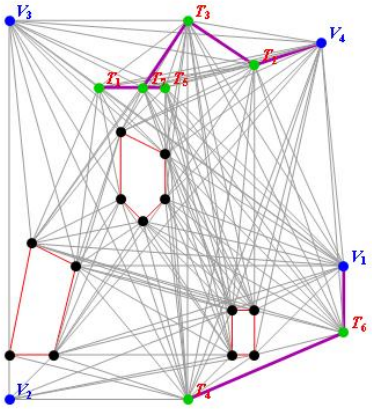

Figure 5.18: Hungarian algorithm solution (4 vehicles, 7 targets)

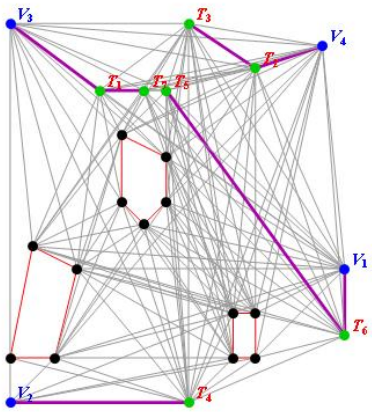

Figure 5.19: Auctioning solution (4 vehicles, 7 targets)

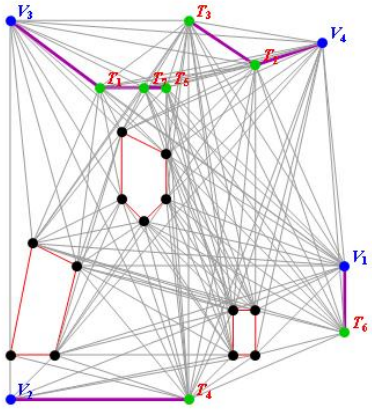

Figure 5.20: Clustering solution (4 vehicles, 7 targets)

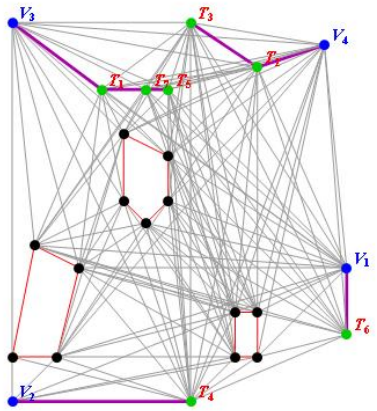

Figure 5.21: DTR solution (4 vehicles, 7 targets)

	J_{time}	J_{fuel}	J_{fuel}-err	J_{dist}
Int. Prog.	12.9301	23.5459	0. %	0.455253
Hung.	14.3734	24.9892	6.12962 %	0.434643
Auct.	16.6015	38.3693	62.9555 %	0.5778
Clust. Hung.	8	25.7678	9.43668 %	0.805245
DTR	8	25.7678	9.43668 %	0.805245

Table 5.6: Scenario results (4 vehicles, 7 targets)

Chapter 6

Conclusions

The problem of cooperative control for a team of unmanned autonomous vehicles has been addressed. Starting from the existing optimal and sub-optimal procedures of path-planning and task assignment some new dynamic approaches have been proposed, focusing on the required computational load.

The proposed path-planning technique is based on an incremental version of the (*Constrained Delaunay Triangulation*, CDT) and on the geometric properties of the triangles. This procedure uses the adjacency information of the incremental version of the CDT and hence it is dynamic. The resulting path is not optimal but it is surely obstacle-free and hence it rapidly provides a safe path among the obstacles of the scenario. A reduction step can be employed online taking advantage again of the geometric properties of the adjacent triangles, thus resulting in a very low computational cost. Simulations have shown the capabilities of the proposed procedure with respect to the other existing procedures, especially for the

computational load that is sensibly lower than the other procedures.

In the context of task-assignment a tasks (targets) clustering approach has been proposed to the end of obtaining a less myopic assignment procedure. The proposed algorithm is dynamic and it automatically determines the *right* number of clusters depending on the value of a tuning parameter ρ. The assignment resulting from the application of the clustering and an assignment technique (such as the *Hungarian Algorithm*, or the *Auctioning*) shows that the final assignment takes care of the *proximity* of the tasks (or the targets) as it was predicted. By this way *close* tasks (targets) can be accomplished by the same vehicle.

Finally a new decentralized and dynamic task-assignment procedure has been proposed. This technique is such that each vehicle builds a *ranking* of the tasks it can serve. Then, negotiating with the other vehicles, it updates its ranking following a nonlinear but simple dynamic. This procedure is proven to be stable and simulations have shown that there exist an unique equilibrium point and it is asymptotically stable. The discretization of such technique, together with the hardware limitations, allows to establish the maximum admissible number of targets and tasks in the scenario depending on the chosen sample time. This is a very interesting property since it gives *a priori* many information about the capabilities of a real-time implementation.

Bibliography

[1] K. Passino, M. Polycarpou, D. Jacques, M. Pachter, Y. Liu, Y. Yang, M. Flint, and M. Baum. *Cooperative Control for Autonomous Vehicle*, chapter 1. Proceedings of the Cooperative Control Workshop, FL, December, 2000.

[2] P. R. Chandler, M. Pachter, D. Swaroop, J. Fowler, J. K. Howlett, S. Rasmussen, C. Schumacher, and K. Nygard. Complexity in uav cooperative control. *Proceedings of the American Control Conference, Anchorage, AK May 8–10, 2002*, 2002.

[3] A. Ryan, M. Zennaro, A. Howell, R. Sengupta, and J. K. Hedrick. An overview of emerging results in cooperative uav control. *Proceedings of the 43rd IEEE Conference on Decision and Control, December 14–17, 2004, Atlantis, Paradise Island, Bahamas*, 2004.

[4] M. Flint, M. Polycarpou, and E. F. Gaucherand. Cooperative control for multiple autonomous uav's searching for targets. *Proceedings of the 41st IEEE Conference on Decision and Control, Las Vegas, Nevada USA, December 2002*, 2002.

[5] D. Enns, D. Bugajski, and S. Pratt. Guidance and control for cooperative search. *Proceedings of the American Control Conference, May 8 - 10, 2002, Anchorage, AK*, 2002.

[6] M. Flint, E. F. Gaucherand, and M. Polycarpou. Cooperative control for multiple autonomous uav's searching risky environments for targets. *Proceedings of the 42nd IEEE Conference on Decision and Control, Maui, Hawaii USA, December 2003*, 2003.

[7] R. W. Beard and T. W. McLain. Multiple uav coopeartive search under collision avoidance and limited range communication constraints. *Proceedings of the 42nd IEEE Conference on Decision and Control, Maui, Hawaii USA, December 2003*, 2003.

[8] S. Li, J. Boskovic, and S. Seereeram anmd R. W. Beard. Autonomous hierarchical control of multiple unmanned compat air vehicles. *Proceedings of the American Control Conference, May 8 - 10, 2002, Anchorage, AK*, 2002.

[9] S. Kapoor and S. N. Maheshwari. Efficient algorithms for euclidean shortest path and visibility problems with polygonal obstacles. In *Symposium on Computational Geometry*, pages 172–182, 1988.

[10] J. Hershberger and S. Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal of Computation*, 28(6):2215–2256, 1999.

BIBLIOGRAPHY

[11] C. W. Warren. gloabl path planning using artificial potential fields. *Proceedings of the IEEE International Conference on Robotics and Automation, 14–19 May 1989*, 1:316–321, 1989.

[12] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[13] H-P. Huang and S-Y. Chung. Dynamic visibility graph for path planning. *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, September 28 - October 2, 2004, Sendai, Japan*, 2004.

[14] T. Pendragon and L. While. Path-planning by tessellation of obstacles. In *ACSC '03: Proceedings of the 26th Australasian computer science conference*, pages 3–9, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[15] L. P. Chew. Constrained delaunay triangulations. In *SCG '87: Proceedings of the third annual symposium on Computational geometry*, pages 215–222, New York, NY, USA, 1987. ACM.

[16] M. V. Anglada. An improved incremental algorithm for constructing restricted delaunay triangulations. *Computers and Graphics*, 21(2):215–223, 1997.

[17] M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained delaunay triangulations. *Geometric Modelling for Scientific Visualization, Heidelberg, Germany*, 2003.

[18] M. Kallmann. Path planning in triangulations. *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games, IJCAI, Edinburgh, Scotland, July 31, 2005*, pages 49–54, 2005.

[19] K. Savla, E. Frazzoli, and F. Bullo. On the point-to-point and traveling salesperson problems for dubins' vehicle. *Proceedings of the American Control Conference, Portland, OR, June 2005*, pages 786–791, 2005.

[20] K. Savla, E. Frazzoli, and F. Bullo. Traveling salesperson problems for the dubins vehicle. *To appear in the IEEE Transactions on Automatic Control*, 2007.

[21] L. Pallottino, V. G. Scordio, and A. Bicchi. Decentralized cooperative conflict resolution among multiple autonomous mobile agents. *Proceedings of the 43rd IEEE Conference on Decision and Control, December 14–17, 2004, Atlantis, Paradise Island, Bahamas*, 2004.

[22] L. Pallottino, V. Scordio, A. Bicchi, and E. Frazzoli. Decentralized cooperative policy for conflict resolution in multivehicle systems. *IEEE Transactions on Robotics and Automation, December 2007*, 23(6):1170–1183, 2007.

[23] S. Rasmussen, P. Chandler, J. W. Mitchell, C. Schumacher, and A. Sparks. Optimal vs. heuristic assignment od cooperative autonomous unmanned air vehicles. *AIAA guidance, Navigation and Control Conference and Exhibit. 11–14 August 2003, Austin, Texas*, 2003.

BIBLIOGRAPHY

[24] D. Turra, L. Pollini, and M. Innocenti. Fast unmanned vehicles task allocation with moving targets. *Proceedings of the 43rd IEEE Conference on Decision and Control, December 14–17, 2004, Atlantis, Paradise Island, Bahamas*, 2004.

[25] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[26] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of SIAM*, 5(1):32–38, 1957.

[27] D. P. Bertsekas and D. A. Castanon. Parallel asynchronous hungarian methods for the assignment problem. *ORSA Journal on Computing*, 5:261–274, 1993.

[28] A. Ahmed, A. Patel, T. Brown, M. Ham, M. Jang, and G. Agha. Task assignment for a physical agent team via a dynamic forward/reverse auction mechanism. *International Conference of Integration of Knowledge Intensive Multi-Agent Systems KIMAS '05: Modeling, Evolutions and Engineering, 18 - 21 April*, 2005.

[29] J. How, T. Shouwenaars, B. De Moor, and E. Feron. Mixed integer linear programming for multi-vehicle path planning. *European Control Conference 2001, Porto, Portugal*, pages 2603–2608, 2001.

[30] J. How, E. King, and Y. Kuwata. Flight demonstrations of cooperative control for uav teams. *AIAA 3rd Unmanned Unlimited Technical Conference, Workshop and Exhibit. 20-23 September 2004, Chicago, Illinois*, 2004.

[31] Y. Kuwata and J. How. Receding horizon implementation of milp for vehicle guidance. *Proceedings of the American Control Conference, Portland, OR, June 2005*, pages 2684–2685, 2005.

[32] A. Bicchi and L. Pallottino. On optimal cooperative conflict resolution for air traffic management systems. *IEEE Transactions on Intelligent Transportation Systems*, 1(4):221–231, 2000.

[33] L. Pallottino, E. M. Feron, and A. Bicchi. Conflict resolution problems for air traffic management systems solved with mixed integer programming. *IEEE Transactions on Intelligent Transportation Systems*, 3(1):3–11, 2002.

[34] L. Pollini, F. Giulietti, and M. Innocenti. Robustness to communication failures within formation flight. *Proceedings of the American Control Conference, 2002*, 4:2860–2866, 2002.

[35] M. M. Polycarpou, Y. Yang, and K. M. Passino. A cooperative search framework for distributed agents. *Proceedings of the 2001 IEEE International Symposium on Intelligent Control, September 5 - 7, 2001, Mexico City, Mexico*, 2001.

[36] M. G. Earl and R. D'Andrea. Iterative milp methods for vehicle-control problems. *IEEE Transactions on Robotics*, 45(6):1158–1167, 2005.

[37] S. Suri and J. O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 14–23, New York, NY, USA, 1986. ACM.

BIBLIOGRAPHY

[38] M. J. Golin. Bipartite matching and the hungarian method. 2006. Course Notes, Hong Kong University of Science and Technology.

[39] T. D. Gottschalk. Concurrent implementation of munkres algorithm. *Proceedings of the Fifth IEEE International Conference on Distributed Memory Computing*, pages 52–57, 1990.

[40] D. P. Bertsekas, D. A. Castanon, and H. Tsaknakis. Reverse auction and the solution of inequality constrained assignment problems. *SIAM Journal on Optimization*, 3:268–299, 1993.

[41] D. P. Bertsekas and D. A. Castanon. A forward/reverse auction algorithm for asymmetric assignment problems. *Technical Report Lids-P-2159, MIT*, 1993.

[42] C. Rosenberger and K. Chehdi. Unsupervised clustering method with optimal estimation of the number of clusters: Application to image segmentation. In *ICPR*, pages 1656–1659, 2000.

[43] C. Moler and C. Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

Appendix A

Software

A.1 F-22 Formation Flight SDRE Controller

This package performs the SDRE controller for an F-22 Formation Flight.

A.1.1 Getting Started

1. Copy all the files into a folder (named BASIC_DIRECTORY).

2. Open one of the `*.mdl` files.

3. *Double-click* on the `Load Data` button. This function sets the initialization variables of the model.

4. Run the simulation.

Base Models The two Simulink models `AFFCSIMv7.mdl` and `AFFCSIMv7_without_VR.mdl` are the reference models without SDRE controller. The former is the original one, provided by *WVU*, the latter performs the same simulation without the Virtual Reality interface.

Hybrid NLDI-SDRE Controller The Simulink model `AFFCSIMv7_SDRE_no_trim.mdl` simulates the SDRE controller of the inner-loop dynamics while the outer-loop is a NonLinear Dynamic Inversion (NDLI) controller.

Full SDRE Controller The Simulink models

- `AFFCSIMv7_SDRE_no_trim_Outer_Controller.mdl`
- `AFFCSIMv7_SDRE_no_trim_xyh_loop.mdl`

simulate the full SDRE controller. This models still have many bugs.

A.1.2 Customize the Software

- The SDC parametrization of the system is realized in the files
  - `FF07_A.m`
  - `FF07_B.m`
  - `FF07_A_OuterLoop.m`
  - `FF07_B_OuterLoop.m`.
- The weight matrices of the SDRE controller are calculated in
  - `FF07_Q.m`
  - `FF07_R.m`

  - `FF07_Q_OuterLoop.m`
  - `FF07_R_OuterLoop.m`

- In order to change the system behavior, modify the weight matrices.

A.2 Cooperative Control GUI

This GUI is not fully working. The not mentioned buttons are not working.

A.2.1 GUI Initialization

1. Copy all the files in a folder (BASIC_DIRECTORY)

2. Change MATLAB current directory to `Cooperative_Control` (named COOPERATIVE_DIRECTORY).

3. Open `initialization.m` and modify the global variable **basic_directory** to COOPERATIVE_DIRECTORY if needed. The default value of **basic_directory** is set to `cd` that is the *current directory*. Typically there is no need to change the value of **basic_directory**.

4. Type `initialization` and press `Enter` on the MATLAB command window. This file sets all the required folders in the MATLAB path. The folders are:

   - BASIC_DIRECTORY/CooperativeGUI

- BASIC_DIRECTORY/Dijkstra
- BASIC_DIRECTORY/Assignment
- BASIC_DIRECTORY/Path_Planning
- BASIC_DIRECTORY/Path_Planning/visibility_graph
- BASIC_DIRECTORY/Path_Planning/delaunay
- BASIC_DIRECTORY/triangle
- BASIC_DIRECTORY/Assignment/Auction
- BASIC_DIRECTORY/Clustering

5. Type `cooperative_gui` and press `Enter` on the MATLAB command window. The GUI will be opened as in figure A.1. The white window is called the *scenario window*

A.2.2 Add Items

This section shows how to add the vehicles, the targets and the obstacles in the scenario. A workspace variable named **scenario** will be created and updated by the GUI. This variable is not intended to be manually modified. However it can be used as a generic MATLAB variable.

Add a Vehicle

- Click on the `New Vehicle` button and move the mouse pointer on the *scenario window* in the GUI.
- A cross indicating the current mouse position will be displayed.

A.2 Cooperative Control GUI

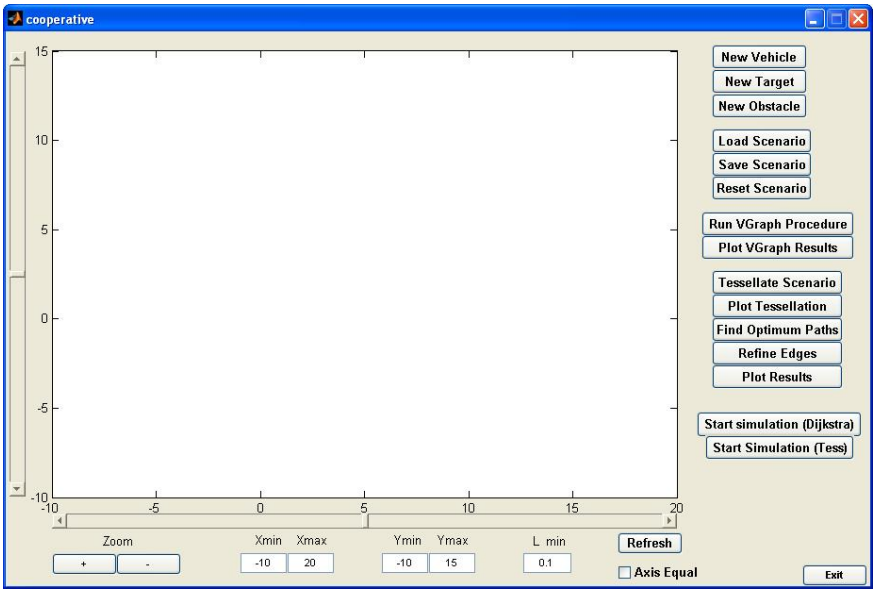

Figure A.1: Cooperative GUI

- Click on the desired vehicle position.
- The cross will be deleted and a blue rectangle will be added. The rectangle identifies the new vehicle position.

Add a Target

- Click on the `New Target` button and move the mouse pointer on the *scenario window* in the GUI.
- A cross indicating the current mouse position will be displayed.
- Click on the desired vehicle position.
- The cross will be deleted and a green circle will be added. The circle identifies the new vehicle position.

Add an Obstacle

- Click on the `New Obstacle` button and move the mouse pointier on the *scenario window* in the GUI.
- A cross indicating the current mouse position will be displayed.
- Click many times on the desired vehicle position to define the obstacles vertices. The added point will not be displayed until the end of the procedure.
- Type `Enter` and the procedure ends.
- The cross will be deleted and a red polygon will be added. The polygon identifies the new obstacle.

A.2.3 Set the Simulation Options

Open the file `set_options.m`. This file sets the simulation options in the workspace variable **opt**:

- `opt.ax = [xmin xmax ymin ymax]`. The four variables represent the corners of the plot region of the simulation.
- `opt.pp_method = 'delaunay'`. The `pp_method` string represents the path-planning method. It can be one of the following:
  - *vgraph* Visibility Graph procedure
  - *delaunay* Constrained Delaunay Triangulation with Adjacency Path.
- `opt.mp_method = 'hungarian'`. The `mp_method` string represents the mission-planning (task-assignment) method. It can be one of the following:
  - *hungarian* Hungarian algorithm.
  - *auction* Auctioning procedure.
  - *clust_hung* Clustering approach with Hungarian algorithm.
  - *clust_auction* Clustering approach with Auctioning procedure.
- `opt.t_end = 11`. Simulation end time.
- `opt.t_step = .025`. Simulation time-step.
- `opt.max_velocity = 2.5`. Maximum vehicles velocity.

- `opt.max_acc = 250`. Maximum vehicles acceleration.
- `opt.obs_traj_smooth = obs_traj_smooth`. Not working.
- `opt.target_traj_smooth = target_traj_smooth`. Not working.
- `opt.cluster_radius = 1`. Cluster radius (parameter ρ).
- `opt.initial_velocity.v = []`. Vehicles initial velocity. If this variable is empty the initial velocity is set to the maximum velocity.
- `opt.T_CONTROLLER = 5`. The number of time-steps after which the cooperative controller is applied. This variable allows to simulate the fact that the cooperative controller has a slower sampling time w.r.t. the system dynamics.

A.2.4 Run a Simulation

IMPORTANT: do not forget to run `set_options.m` before each simulation.

In order to run the simulation, type `play_scenario(scenario,opt)` and then press `Enter` in the MATLAB command window. A plot window will be opened and the simulation stars.

The function `play_scenario` admits two optional outputs:

vehicles A structure of length `n` (where `n` is the total number of vehicles in the scenario) containing the following two fields:

- `velocity`: the constant speed of the vehicle

- `position`: the position of the vehicle at the different sample times.

MOV A `movie` variable. Once the simulations is finished, the movie can be viewed typing `movie(MOV)` and then pressing `Enter`.

A.2.5 How to Customize the Software

Add a New Path-Planning Method

The realized path planning methods are contained in the file `path_planning_method.m`. The following procedure explains how to add a custom path-planning method:

1. Choose an identification string of the custom path-planning method (e.g. *my_pp_method*)
2. Go to the end of the file `path_planning_method.m` and add the following lines before the last `else` statement:

```
elseif strcmp(pp_method, 'my_pp_method')

{

% Add here the custom implementation.

}
```

3. The string `'my_pp_method'` is now recognized by the software as a new path-planning method.

Add a New Task-Assignment Method

The realized path planning methods are contained in the file `path_planning_method.m`. The following procedure explains how to add a custom path-planning method:

1. Choose an identification string of the custom path-planning method (e.g. *my_mp_method*)
2. Go to the end of the file `assignment_evaluation.m` and add the following lines before the last `else` statement:

```
elseif strcmp(mp_method, 'my_mp_method')
{
% Add here the custom implementation.
}
```

3. The string `'my_mp_method'` is now recognized by the software as a new task-assignment method.

A.3 CCTool - Cooperative Control Simulink Tool

CCTool is a Simulink toolbox for simulating cooperating UAVs in a dynamic scenario. The tool is composed by a set of different blocks

performing different actions. Whenever necessary, the blocks are implemented in a decentralized manner, so that multiple block instances can be used.

A.3.1 Initialization

1. Copy all the files into a directory (named BASIC_DIRECTORY).

2. Open the file `init_cctool.m` and modify the variable **basic_directory** to BASIC_DIRECTORY if needed. The default value of **basic_directory** is set to `cd` that is the *current directory*. Typically there is no need to change the value of **basic_directory**.

3. Run `init_cctool`. This file sets the necessary folders into the MATLAB path.

4. The tool is now ready to be used.

A.3.2 Using the Package

CCTool is not currently a full package and it only works on many examples (*.mdl files). In the following the different block types are described. If not explicitly written, it is intended that all the blocks are decentralized and hence the prefix `decentralized_` must be added to every block name. In addition, since all the blocks are realized as MATLAB *s-functions* then the suffix `_sfun` must be added.

add_moving_obstacle

This function adds an obstacle to the scenario. The obstacle is characterized as an octagon whose center and radius are specified by the corresponding inputs. The obstacle TYPE reflect the fact that the obstacle is *seen* by an aerial, ground or underwater vehicle.

- Inputs
  1. Obstacles: a $4 \times nMax$ matrix containing the obstacles
  2. Center: a 2×1 vector containing the
  3. Radius: a scalar value containing the obstacle radius
  4. ObstacleID: a scalar and integer value containing the obstacle identification number.
  5. ObstacleTYPE: a scalar and integer value containing the obstacle TYPE (1 = aerial, 0 = ground, -1 = underwater)
- Outputs
  1. NewObstacle: a $4 \times nMax$ matrix containing the new obstacle structure.

benefit_evaluation

This function evaluates the benefits from a given costs vector. The benefits are used in the *DTR* task assignment procedure.

- Inputs
  1. Tasks cost: a $nMax \times 1$ vector containing the costs of the tasks.

2. Tasks ID: a $nMax \times 1$ vector containing the tasks identification numbers.

- Outputs

1. Benefits: a $2 \times nMax$ matrix containing the tasks benefits in the first row, and the tasks identification numbers in the second row.

build_cost_matrix

This function builds the cost matrix in order to use it into an assignment procedure.

- Inputs

1. Global cost matrix: a $nMax \times nMax$ matrix containing all the costs between each vehicle and each node of the visibility graph. In the case of a decentralized system the number of vehicles must be one.
2. ID_start: a $2 \times nMax$ matrix containing the identification number of the vehicles.
3. ID_end: a $2 \times nMax$ matrix containing the identification number of the targets.

- Outputs

1. Cost matrix: a $nMax \times nMax$ matrix containing the cost between each vehicle and each task.

choose_task

This function is used in the DTR assignment procedure and chooses the best task for the given vehicle.

- Inputs
  1. Weights: a $nMax \times 1$ vector containing the actual weights value of the vehicle.
  2. Tasks ID: a $nMax \times 1$ vector containing the tasks identification numbers.
- Outputs
  1. Chosen task: an integer scalar value containing the chosen task.

collect_bxw

This function collects the Bw products coming from the other vehicles (for the use with DTR).

- Inputs
  1. A $nMax \times 2nMax$ matrix containing the Bw values and the respective task identification number coming from the other vehicles. An extern vehicle is identified with two columns: the first contains the Bw values, the second contains the tasks identification numbers.
- Outputs

1. A $nMax \times (nMax + 1)$ matrix containing the total sum of the Bw products. The last columns contains the tasks identification numbers.

create_adjacency_graph

This function creates the adjacency graph from the coordinates of the incenters of the triangles and the adjacency information coming from the dynamic CDT algorithm.

- Inputs

  1. A $3 \times nMax$ matrix containing the coordinates of the incenters of the triangles.
  2. A $nMax \times nMax$ matrix containing the adjacency information. If the (i, j) entry is 0 then the triangles i and j are not adjacent, otherwise these are adjacent.

- Outputs

  1. A $nMax \times nMax$ matrix containing the adjacency graph cost matrix. If two points (incenters) are not directly linked then the associated cost is set to -1.

decode_task

This function extracts the next waypoint the vehicle must reach.

- Inputs

1. A $2 \times nMax$ matrix containing the start points identification numbers in the first row, and the vehicles identification numbers in the second row. If the function is used in a decentralized system the number of vehicles must be one.
2. A $2 \times nMax$ matrix containing the end points identification numbers in the first row, and the targets identification numbers in the second row.
3. A $nMax \times nMax$ matrix containing the predecessors matrix (see the Dijkstra Algorithm for the details).
4. An integer scalar value containing the vehicle identification number.
5. An integer scalar value containing the selected task identification number.

- Outputs

1. An integer scalar value containing the next waypoint identification number.

dijkstra

This function performs the Dijsktra algorithm.

- Inputs

1. A $nMax \times nMax$ matrix containing the cost between each node to any other node.

2. A $2 \times nMax$ matrix containing the start nodes in the first row, and the respective identification numbers in the second row.

- Outputs

1. A $nMax \times nMax$ matrix containing the predecessors of each node.
2. A $nMax \times nMax$ matrix containing the path cost between the start nodes and each other node.

dynamic_cdt

This function performs the dynamic *Constrained Delaunay Triangulation*.

- Inputs

1. A $3 \times nMax$ matrix containing the nodes to be triangulated. The last row contains the nodes identification number.
2. A $2 \times nMax$ matrix containing the identification numbers of the constrained edges. The two rows contain the identification number of the first and the second node of each constraint respectively.

- Outputs

1. A $3 \times nMax$ matrix containing the vertices identification number of the built triangles.

2. A $2 \times nMax$ matrix containing the nodes coordinates (one node for each column).
3. A $nMax \times nMax$ matrix containing the first adjacency information. Each entry (i, j) contains the identification number of the triangle adjacent to the edge (i, j).
4. Similar to the previous output (there are at most two adjacent triangles).
5. A $nMax \times nMax$ matrix containing the fixed edges of the triangulation.

find_target

This function finds the target xyz-coordinates from a given identification number.

- Inputs
  1. A $4 \times nMax$ matrix containing the *scenario*.
  2. An integer scalar value containing the target identification number.
- Outputs
  1. A 3×1 vector containing the target position. The last entry of the vector represents its altitude and it is set to zero.

find_waypoint

This function finds the waypoint xyz-coordinates from a given waypoint-identification number.

- Inputs
  1. A $3 \times nMax$ matrix containing the nodes of the scenario.
  2. An integer scalar value containing the waypoint identification number.
- Outputs
  1. A 3×1 vector containing the chosen waypoint coordinates. The last entry represents the waypoint altitude and it is set to zero.

generate_vehicle_reference

This function calculates the reference position for a the vehicle (for the use with the dynamic CDT with adjacency path).

- Inputs
  1. A 3×1 vector containing the vehicle position.
  2. A $nMax \times nMax$ matrix containing the predecessors matrix.
  3. A 2×1 vector containing the target: the first row contains the node (associated to the target) identification number while the second row contains the target identification number.

4. A $3 \times nMax$ matrix containing all the possible waypoints.
5. A $nMax \times nMax$ matrix containing the first adjacency information.
6. Similar to the previous input, containing the second adjacency information.
7. A $2 \times nMax$ matrix containing the triangles resulting from the dynamic CDT.
8. A 3×1 vector containing the target xyz-coordinates.
9. A $2 \times nMax$ matrix containing the start nodes identification number.
10. A $2 \times nMax$ matrix containing the end nodes identification number.
11. An integer scalar value containing the vehicle identification number.

- Outputs

1. A 3×1 vector containing the reference-point of the vehicle.

get_wp_coords

This function finds the waypoint xyz-coordinates from a given waypoint-identification number.

- Inputs

1. A $3 \times nMax$ matrix containing all the waypoints coordinates.

2. An integer scalar value containing the waypoint identification number.

- Outputs

1. A $3 \times nMax$ vector containing the waypoint xyz-coordinates.

incenters

This function finds the incenters of the triangles resulting from the dynamic CDT.

- Inputs

1. A $3 \times nMax$ matrix containing the triangles resulting from the dynamic CDT.
2. A $2 \times nMax$ matrix containing the nodes of the triangulation.
3. A $4 \times nMax$ matrix containing the *scenario*.
4. A $nMax \times nMax$ matrix containing the first adjacency information.
5. Similar to the previous input. Contains the second adyacency information.
6. A 2×4 matrix containing the scenario bounds.

- Outputs

1. A $3 \times nMax$ matrix containing the incenters coordinates.
2. A $nMax \times nMax$ containing the updated first adjacency information (the not-admissible triangles are deleted).

3. Similar to the previous output. Contains the updated second adjacency information.
4. A $2 \times nMax$ matrix containing the start nodes identification number.
5. A $2 \times nMax$ matrix containing the end nodes identification number.

set_cdt_inputs

This function sets the right data structure for the use with the dynamic CDT function.

- Inputs
  1. A $4 \times nMax$ matrix containing the *scenario*.
  2. A 2×4 matrix containing the scenario bounds.
- Outputs
  1. A $3 \times nMax$ matrix containing the modified scenario (the vehicles and the targets are deleted).
  2. A $2 \times nMax$ matrix containing the CDT constraints.

set_scenario

This function sets the scenario structure.

- Inputs
  1. A 3×1 vector containing the vehicle xyz-coordinates.

2. A $4 \times nMaxScenario$ matrix containing the targets coordinates and identification number.
3. A $4 \times nMaxScenario$ matrix containing the obstacles coordinates and identification number.
4. An integer scalar value containing the vehicle identification number.
5. An integer scalar value containing the vehicle type (aerial, ground or underwater).

- Outputs

1. A $4 \times nMax$ matrix containing the *scenario*.

set_tasks_cost

This function calculates the tasks cost for the vehicle.

- Inputs

1. A $2 \times nMax$ matrix containing the start nodes identification number.
2. A $2 \times nMax$ matrix containing the end nodes identification number.
3. A $nMax \times nMax$ matrix containing all the task cost.
4. An integer scalar value containing the vehicle identification number.

- Outputs

1. A $nMax \times 1$ vector containing the tasks cost.
2. A $nMax \times 1$ vector containing the tasks identification number.

set_vgraph_inputs

This function sets the right data structure for the use with the *Visibility Graph* procedure.

- Inputs
  1. A $4 \times nMax$ matrix containing the *scenario*.
- Outputs
  1. A $3 \times nMax$ matrix containing the nodes of the visibility graph.
  2. A $2 \times nMax$ matrix containing the constraints of the visibility graph.
  3. A 1×3 matrix containing the number of vehicles, targets and obstacles respectively.
  4. A $2 \times nMax$ matrix containing the start nodes identification number.
  5. A $2 \times nMax$ matrix containing the end nodes identification number.

targets_management

This function simulates the interaction between the vehicles and teh targets. It simulates the events of *visit a target*.

- Inputs
  1. A $4 \times nMax$ matrix containing the targets coordinates (first two rows) together with their identification numbers (third row) and type (fourth row).
  2. A $3 \times nMax$ matrix containing the vehicles coordinates.
- Outputs
  1. A $4 \times nMax$ matrix containing the updated targets (with the same meaning of the first input).

vgraph_sfun

This function builds the *visibility graph*.

- Inputs
  1. A $3 \times nMax$ matrix containing the nodes of the visibility graph.
  2. A $2 \times nMax$ matrix containing the constraints of the visibility graph.
  3. A 1×3 matrix containing the number of vehicles, targets and obstacles respectively.
- Outputs
  1. A $nMax \times nMax$ matrix containing the node-to-node cost of the visibility graph.

weight_dynamics

This function performs the DTR dynamics.

- Inputs
  1. A $2 \times nMax$ matrix containing the benefits and the identification numbers of the tasks.
  2. A $nMax \times (nMax+1)$ matrix containing the Bw product of of the other vehicles. The last column contains the tasks identification number.
  3. A $nMax \times 1$ vector containing the initial values of the weights.
- Outputs
  1. A $nMax \times 1$ vector containing the weights.
  2. A $nMax \times 1$ vector containing the Bw product sent to the other vehicles.
  3. A $nMax \times 1$ vector containing the tasks identification numbers.

A.3.3 La Spezia Scenario

In this section the main steps to run the simulation of the *La Spezia* scenario are presented.

1. Initialize CCTool as explained before.
2. Change current directory to `Matlab_LaSpezia`.

3. Run `LaSpeziaInitialization`

4. Open one of the `*.mdl` files in the folder and run the simulation.